ÍTALO CAMPOS DE MELO SILVA

TEMPORIZADORES EM SOFTWARE PARA LINUX DE TEMPO REAL: UMA PROPOSTA PARA DIMINUIR INTERFERÊNCIAS EM PROCESSOS DE TEMPO REAL.

FLORIANÓPOLIS
2010

UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

TEMPORIZADORES EM SOFTWARE PARA LINUX DE TEMPO REAL: UMA PROPOSTA PARA DIMINUIR INTERFERÊNCIAS EM PROCESSOS DE TEMPO REAL.

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia
de Automação e Sistemas.

ÍTALO CAMPOS DE MELO SILVA

Florianópolis, Agosto de 2010.

Catalogação na fonte pela Biblioteca Universitária
da
Universidade Federal de Santa Catarina

S586t Silva, Ítalo Campos de Melo
Temporizadores em software para linux de tempo real [dissertação] : uma proposta para diminuir interferências em processos de tempo real / Ítalo Campos de Melo Silva ; orientador, Rômulo Silva de Oliveira. - Florianópolis, SC, 2010.
270 p.: il., grafs., tabs.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia de Automação e Sistemas.

Inclui referências

1. Engenharia de sistemas. 2. Temporizadores em software. 3. Sistema operacional. 4. Tempo real. I. Oliveira, Rômulo Silva de. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Automação e Sistemas.. III. Título.

CDU 621.3-231.2(021)

TEMPORIZADORES EM SOFTWARE PARA LINUX DE TEMPO REAL: UMA PROPOSTA PARA DIMINUIR INTERFERÊNCIAS EM PROCESSOS DE TEMPO REAL.

Ítalo Campos de Melo Silva

'Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia de Automação e Sistemas, Área de Concentração em *Controle, Automação e Sistemas*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia de Automação e Sistemas da Universidade Federal de Santa Catarina.'

Rômulo Silva de Oliveira, Dr.
Orientador

Luciano Porto Barreto, Dr.
Co-orientador

José Eduardo Ribeiro Cury, Dr.
Coordenador do Programa de Pós-Graduação
em Engenharia de Automação e Sistemas

Banca Examinadora:

Rômulo Silva de Oliveira, Dr.
Presidente

Luciano Porto Barreto, Dr.

Antônio Augusto Medeiros Fröhlich, Dr.

Carlos Barros Montez, Dr.

Lau Cheuk Lung, Dr.

Ao meu pai e meu avô in memorian.

A minha mãe e minha avó.

Agradecimentos

Agradeço primeiramente a Deus por tudo o que ele tem proporcionado de bom em minha vida e por nunca ter me abandonado em momento algum.

Agradeço ao meu avô e meu pai *in memorian*, por sempre terem me apoiado nos estudos. A minha avó que sempre me incentivou a tentar fazer tudo e por sempre acreditar em mim. A minha mãe que também sempre me apoiou nos estudos.

Agradeço ao meu professor e orientador Rômulo Silva de Oliveira e meu co-orientador Luciano Porto Barreto, por seus conselhos e orientação.

Agradeço muito a todos os professores do DAS por todo o conhecimento que me passaram.

Agradeço aos meus amigos Daniel Kullkamp, Daniel Mayer, Denis, Giovani, Jim Lau, José Vergara, Mateus, Rodrigo Lange, Tanísia, Vitor e Yuri por me ajudarem de várias maneiras durante dois anos, como também por toda a amizade e momentos de descontração. Agradeço especialmente ao meu amigo Andreu Carminati, por toda a ajuda durante o desenvolvimento da dissertação. Também agradeço a todos aqueles que de alguma forma contribuíram para a realização deste trabalho.

Agradeço a Universidade Federal do Acre por ter permitido meu afastamento para este programa de pós-graduação.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia de Automação e Sistemas.

TEMPORIZADORES EM SOFTWARE PARA LINUX DE TEMPO REAL: UMA PROPOSTA PARA DIMINUIR INTERFERÊNCIAS EM PROCESSOS DE TEMPO REAL.

Ítalo Campos de Melo Silva

Agosto/2010

Orientador: Rômulo Silva de Oliveira, Dr.
Co-orientador: Luciano Porto Barreto, Dr.
Área de Concentração: Controle, Automação e Sistemas.
Palavras-chave: Temporizadores em Software, Sistema Operacional, Tempo Real.
Número de Páginas: xxiv + 94

Em sistemas de tempo real, as tarefas devem executar em um período de tempo previsível e sem atrasos, para assim garantir o bom funcionamento do sistema. Este trabalho trata sobre o que ocorre no Linux com a utilização do pacote de tempo real PREEMPT-RT. O problema encontrado é uma inversão de prioridades que os processos de tempo real sofrem, a qual ocorre através da execução de alguns temporizadores de alta resolução, mais especificamente, os temporizadores responsáveis por acordar os processos que estavam dormindo por um certo período de tempo. Quando estes processos precisam acordar, os temporizadores preemptam qualquer processo em execução para isto. Neste caso, processos de menor prioridade interferem na execução de processos com maior prioridade. Para resolver este problema, este trabalho propõe a postergação da execução destes temporizadores, os executando em momentos apropriados, de forma que respeitem as prioridades dos processos e não posterguem demais o início da execução dos processos que devem acordar.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Automation and Systems Engineering.

TEMPORIZADORES EM SOFTWARE PARA LINUX DE TEMPO REAL: UMA PROPOSTA PARA DIMINUIR INTERFERÊNCIAS EM PROCESSOS DE TEMPO REAL.

Ítalo Campos de Melo Silva

August/2010

Advisor: Rômulo Silva de Oliveira, Dr.
Luciano Porto Barreto, Dr.
Area of Concentration: Control, Automation and Systems
Key words: Software Timers, Operating System, Real Time.
Number of Pages: xxiv + 94

In real-time systems, tasks must run in a predictable period of time and without delay, thus ensuring the smooth functioning of the system. This paper deals with what happens in Linux using the package of real-time PREEMPT-RT. The problem encountered is a priority inversion that the processes of real-time experience, which occurs through the execution of some high resolution timers, more specifically, timers responsible for waking up processes that were sleeping for a certain period of time. When these processes need to wake up, timers preempt any running process for it. In this case, lower priority processes interfere with the running processes with higher priority. In order to solve this problem, this paper proposes the postponement of the running of these timers, running them at the appropriate times in order to respect the priorities of other processes and not to postpone the start of implementation of procedures that must wake up.

Sumário

Lista de Abreviações

ACPI PMT - Advanced Configuration and Power Interface Power Management Timer

APIC - Advanced Programmable Interrupt Controller

BIOS - Basic Input/Output System

CLK - Clock

CMOS RAM - Complementary Metal-Oxide-Semiconductor Random Access Memory

HPET - High Precision Event Timer

IRQ - Interrupt Request

PC - Personal Computer

PIT - Programmable Interval Timer

RTC - Real Time Clock

TOD - Time of Day

TSC - Time Stamp Counter

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Um sistema computacional moderno é formado por um ou mais processadores, memória principal, discos, impressores, teclado, monitor de saída, *interfaces* de rede e outros dispositivos de entrada ou saída. Gerenciar e controlar todos estes componentes de forma correta, as vezes até otimizada, é uma tarefa complexa. Se todo programador tivesse que se preocupar com o funcionamento das unidades de disco e todos os problemas que pode ocorrer ao se ler um bloco do disco, além da manipulação dos periféricos de entrada e saída, desenvolver um programa se tornaria bem mais complexo do que já é. Devido a isso, há muito tempo tornou-se bastante evidente a necessidade de encontrar uma maneira de distanciar o programador da complexidade do *hardware*. A solução foi colocar uma camada de *software* sobre a de *hardware*, gerenciando assim todas as partes do sistema e apresentando ao usuário uma *interface* mais fácil de entender e programar, a esta camada foi dado o nome de sistema operacional, também conhecido como sistema operacional de propósito geral [44].

Vários sistemas operacionais foram criados de acordo com a evolução do *hardware*, alguns não prosseguiram sua evolução de forma satisfatória, enquanto outros evoluíram para versões mais robustas e melhoradas, tais como Windows, Mac OS, Linux, entre outros. Com a evolução destes sistemas operacionais de propósito geral, começaram a ser desenvolvidos versões de sistemas operacionais para aplicações de tempo real, os quais surgiram da demanda de uma camada de *software* que garantisse maior precisão para processos de tempo real.

Um bom sistema operacional de tempo real não provê apenas mecanismos e serviços suficientes para garantir bom escalonamento de tempo real e políticas de escalonamento de recursos, mas também mantém seu próprio tempo e consumo de recursos previsíveis [29]. Como processos de tempo real precisam executar em um tempo máximo previsto, para assim garantir a realização de suas tarefas no tempo necessário, eles podem ser programados especificamente para um *hardware* (de forma mais complexa), para assim poder planejar e garantir seus tempos de execução, como também podem ser programados sobre um sistema

operacional de tempo real, abstraindo o trabalho do programador de gerenciar os recursos de *hardware* e ainda assim garantir o período de execução destes processos.

Existem muitos processos que necessitam serem executados em contexto de tempo real, tais como processos de aplicação multimídia, os quais precisam geralmente processar muitos dados e apresentá-los ao usuário em uma frequência muito alta, como filmes ou músicas, mas que neste caso a perda de um *deadline* não é tão grave, podendo apenas atrasar alguns *frames* da apresentação. Um exemplo de um sistema de tempo real mais crítico, o qual o não cumprimento de seus deadlines pode acarretar problemas graves, é um sistema de controle de vôo, que caso uma ação importante seja atrasada por algum motivo, pode até ocasionar na queda de um avião [29].

O Linux é um sistema operacional de propósito geral, criado originalmente por Linus Torvalds com a ajuda de desenvolvedores de todo o mundo [37]. Ele é desenvolvido sob a licença pública geral GNU, o que torna o código fonte do seu *kernel* aberto para qualquer estudante, pesquisador, ou apenas curioso [21]. Sua versão padrão implementa algumas características de tempo real, mas existem pacotes que o modificam e o melhoram em relação ao suporte dos processos de tempo real. Neste trabalho em específico, é utilizado o Linux com a versão do *kernel* 2.6.31.6 [40], mas utiliza-se também um pacote para tempo real no Linux conhecido como PREEMPT-RT [32]. Desta forma, pode-se estudar e analisar o código do *kernel* de um sistema operacional de tempo real, tentando contribuir de alguma forma para a otimização ou alteração significativa em alguma parte deste código, o qual está em constante desenvolvimento.

O equipamento utilizado para o desenvolvimento deste trabalho foi um *laptop* com processador Intel Core 2 Duo de 2 GHz, com memória RAM de 2 GB e disco de 250 GB. Este equipamento foi utilizado para a execução do Linux com o PREEMPT-RT, como também para a alteração, compilação e execução do *kernel* do Linux para sua versão modificada, implementando assim a proposta deste trabalho.

1.1 Motivação

Existem alguns fatores que motivaram o desenvolvimento deste trabalho, como o fato do código do *kernel* do Linux ser aberto, possibilitando seu estudo ou alteração, podendo assim analisar todo o conceito de um sistema operacional na prática. Ainda mais por este sistema ser tão completo e utilizado de forma tão ampla pelo mundo todo.

Outro fator motivador deste trabalho foi a possibilidade de analisar como o Linux se comporta utilizando o pacote PREEMPT-RT, por ser um pacote de tempo real que não necessita de nenhuma camada de *hardware* para funcionar e por estar sempre em constante

desenvolvimento. Um campo de estudo interessante dentro de um *kernel* de um sistema operacional de tempo real é sua parte de temporizadores. Os temporizadores são responsáveis por controlar a frequência que o escalonador atua, o tempo que os processos devem dormir, entre outros fatores que direta ou indiretamente podem alterar a precisão do sistema, que é algo importante para sistemas de tempo real. Assim, estudar e entender o funcionamento dos temporizadores para tentar melhorá-los de alguma forma se torna interessante, considerando ainda que o Linux e o pacote de tempo real PREEMPT-RT estão em constante desenvolvimento, podendo conter melhorias e otimizações a serem realizadas, como também até erros ainda não encontrados.

Mais um fator motivador foi um problema encontrado durante o estudo do *kernel* e principalmente dos seus temporizadores de alta resolução. Ocorre que quando um temporizador expira, ele gera uma interrupção que é tratada pelo seu tratador de interrupções, o qual interfere qualquer processo em execução no momento. Entre estes temporizadores existem os que necessitam executar com alta precisão e realmente agem desta forma, preemptando os processos para poderem executar, como também existem aqueles que não necessitam executar com tanta precisão, tendo seu processamento postergado para não causar grande interferência ao processo em execução. Mas entre estes temporizadores utilizados com alta precisão, existem os que são utilizados para acordar processos depois de um período de tempo, geralmente conhecidos como *sleeps*. O problema é que processos de baixa prioridade podem ser acordados enquanto um processo de tempo real de alta prioridade está executando. Desta forma, além do processo que está sendo acordado não poder executar ainda por ter a prioridade menor, o temporizador que o acorda interfere na execução do processo de maior prioridade, ocasionando além de *overhead* devido a interferência, uma inversão de prioridade por meio deste temporizador, pois executa uma função de um processo de baixa prioridade interrompendo a execução de um processo de alta prioridade.

1.2 Objetivos

Depois de encontrar o problema da interferência que o tratador de interrupções dos temporizadores de alta resolução cria, tornou-se objetivo deste trabalho minimizar esta interferência, postergando parte do trabalho deste tratador para um momento oportuno, já que não se pode cancelar a interrupção causada.

Com base no objetivo geral, pode-se citar as seguintes etapas deste trabalho:

- Realizar o levantamento bibliográfico sobre temporizadores do Linux, que é uma parte de grande importância para o Linux em relação a seu suporte de tempo real e também é onde se encontra o problema de inversão de prioridades, que causa a interferência mencionada anteriormente.

- Identificar quais temporizadores de alta resolução fazem parte do problema e devem ser postergados para diminuir a interferência causada.
- Propor as alterações necessárias no *kernel* do Linux estudado com o pacote PREEMPT-RT, para assim resolver a inversão de prioridades identificada e diminuir o tempo de interferência existente com esta inversão.
- Aplicar as alterações propostas no Linux estudado, criando uma versão alterada, servindo como protótipo para a aplicação da proposta.
- Comparar a versão do Linux estudada com a modificada, analisando suas diferenças e tempo de execução.

1.3 Organização do trabalho

Este trabalho está organizado em seis capítulos, dos quais o primeiro trata-se desta introdução. O segundo capítulo trata sobre alguns conceitos básicos necessários para o entendimento e desenvolvimento deste trabalho, tratando sobre processos de um sistema operacional, interrupções do Linux e o escalonamento de tempo real que o Linux estudado com o pacote PREEMPT-RT possui.

O terceiro capítulo é o mais longo, o qual aborda todo o conceito de temporizadores no Linux, desde os temporizadores de *hardware* existentes, até os temporizadores de *software* e como eles são divididos em temporizadores de baixa e de alta resolução.

O quarto capítulo descreve o problema encontrado, detalhando-o e mostrando o porquê ele não deve ocorrer, enquanto que o quinto capítulo descreve a proposta do trabalho, a sua implementação no *kernel* do Linux estudado e a análise comparativa entre o *kernel* normal e a implementação da proposta. O trabalho termina com o sexto capítulo, o qual contém a conclusão do trabalho realizado.

Capítulo 2

Processos, Interrupções e Escalonamento de Tempo Real no Linux

O Linux como todo sistema operacional moderno, pode executar vários processos de forma que o usuário tenha a impressão que eles estejam executando ao mesmo tempo, onde cada processo é formado por um conjunto de instruções que precisam ser realizadas. Para que estes processos sejam executados desta forma, o Linux precisa gerenciá-los de forma rápida e precisa, respeitando algumas regras.

Outra característica muito importante no Linux é a utilização de interrupções na execução do processador, para assim poder mudar o seu foco e alterar seu fluxo de execução, chamando a atenção do processador para algum processo que precise ser executado naquele instante.

Processos e interrupções do processador são características importantes para o entendimento do funcionamento de um sistema operacional. Mas como este trabalho não abrange o estudo completo de um sistema operacional, este capítulo busca explicar brevemente algumas características básicas sobre processos, interrupções e a classe de escalonamento de tempo real para o melhor entendimento do desenvolvimento deste trabalho.

2.1 Prioridades de Processos

Nem todos os processos de um sistema operacional são iguais, alguns são mais importantes que outros, sendo esta importância identificada através de prioridades. Um processo importante pode ter prioridade alta, enquanto outros menos importantes possuem prioridades

menores. Desta forma, o sistema operacional identifica qual processo deve ter sua execução priorizada em relação ao conjunto de processos prontos [22].

Existem diferentes classes de prioridades para satisfazer diferentes demandas de processos, podendo ser divididas em classe de processos normais e de tempo real, onde cada classe tem diferentes formas de organizar e executar seus processos, precisando utilizar regras próprias. Mesmo dentro de algumas classes, ainda pode haver pequenas diferenças no tratamento de alguns processos, como na classe de tempo real, que os processos podem ser divididos em processos de tempo real crítico ou não crítico [22][31].

Os processos de tempo real crítico necessitam respeitar limites de tempo com severidade, completando suas tarefas no tempo correto, caso contrário podem acarretar problemas graves ao sistema. O Linux padrão (*vanilla*) não suporta processamento em tempo real crítico, mas existem versões modificadas do Linux que suportam, como Xenomai [23] ou RTAI [18]. Os processos de tempo real crítico possuem a maior prioridade de todo o sistema, ou seja, quando um processo destes entrar na fila de prontos do processador e não existir nenhum outro da mesma classe com prioridade maior ou igual a dele, ele é processado, preemptando qualquer outro processo que estivesse executando no processador. Neste modelo, o *kernel* do Linux é considerado um processo com uma determinada prioridade, mas todos os processos de tempo real crítico possuem prioridade maior que esta, assim até o *kernel* é preemptado para que os processos mais prioritários sejam executados e consigam garantir seus tempos de execução.

Os processos de tempo real não crítico também precisam respeitar limites de tempo em sua execução, mas de forma não tão severa. Caso a execução de um de seus processos exceda o limite de tempo previsto, sofrerá um pequeno atraso, mas não causará problemas graves ao sistema. Mesmo o Linux padrão não tendo como prioridade o seu desenvolvimento para suportar processos de tempo real, ele possui nativamente implementado o padrão POSIX.4, já suportando assim processos de tempo real não críticos [35] [8].

O *patch* utilizado neste trabalho, chamado PREEMPT-RT, é não intrusivo no sentido de não utilizar nenhum tipo de camada de abstração de *hardware*. Realiza alterações no *kernel* do Linux e o melhora para suportar processos de tempo real não crítico (*soft real-time*), reduzindo a latência de escalonamento através da opção de preempção completa que ele possui, entre outras alterações [8] [25]. Quando alguma característica do pacote PREEMPT-RT se torna estável e interessante para se ter no *kernel* padrão, ele é copiado e utilizado no *kernel* normal, como é o caso dos temporizadores de alta resolução que foram copiados para o *kernel* padrão. Tanto no Linux padrão como com o PREEMPT-RT, os processos de tempo real têm prioridades mais altas do que os processos normais, preemptando qualquer processo normal, sempre que estejam na fila de prontos do processador. Desta forma, eles tentam garantir o limite de tempo em que devem executar.

Os processos normais também são classificados através de prioridades entre os mais ou

menos importantes, só que eles nunca podem ser mais importantes que os processos de tempo real. Estes processos são divididos em fatias de tempo, onde cada processo possui uma fatia de tempo de acordo com sua prioridade. Desta forma, um processo com alta prioridade possui uma fatia de tempo maior para sua execução, enquanto um processo com menor prioridade tem uma fatia de tempo menor. Cada processo executa durante sua fatia de tempo, quando esta acaba é restaurada para seu valor original, ou seja, é zerada e o *kernel* seleciona outro processo para executar da mesma forma, continuando de maneira cíclica entre os processos normais na fila de prontos do processador, até concluírem suas devidas execuções [31].

Em relação as prioridades que os processos podem assumir, existe uma faixa de prioridades para os processos normais e outra para os de tempo real. A faixa dos processos normais está entre -20 e +19, sendo que este valor é inversamente proporcional, pois quanto menor ele for, maior será a prioridade real do processo, ou seja, um processo normal com prioridade -20 é o processo mais prioritário entre os desta classe, enquanto que outro processo desta classe com prioridade 19 é o menos prioritário de todo o sistema. A faixa de prioridades dos processos de tempo real varia de 0 até 99, sendo 0 a menor prioridade desta classe e 99 a maior. Desta forma, um processo de tempo real com prioridade igual a 99 é o processo mais prioritário de todo o sistema [31].

O *kernel* trata internamente todas estas prioridades com uma simples faixa de valores de 0 até 139, onde os valores mais baixos se referem as maiores prioridades, invertendo a forma que o usuário enxerga as prioridades dos processos. Portanto, a faixa interna de 0 até 99 é utilizada para processos de tempo real, que corresponde de forma inversa a faixa de tempo real que o usuário utiliza. A maior prioridade para o usuário é 99 e corresponde ao valor zero que é a maior prioridade internamente, enquanto que o menor valor para o usuário é zero e corresponde internamente ao valor 99. Já os processos normais, cujas prioridades para o usuário podem variar entre -20 e +19, são mapeados internamente pela faixa de valores de 100 até 139, onde -20 se refere internamente ao valor 100 e +19 ao valor 139, como pode ser observado na figura 2.1 [31] [26].

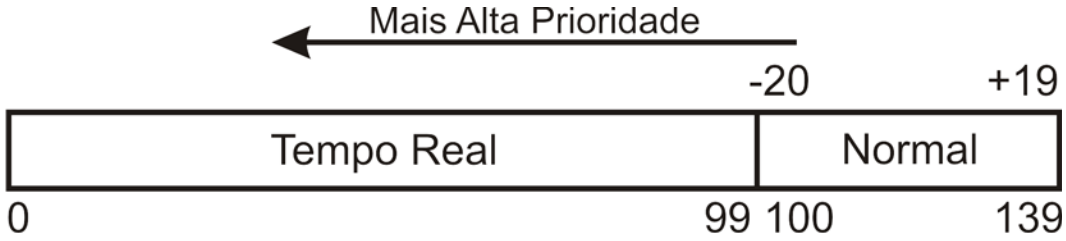

Figura 2.1: Escala de prioridades de processos. [31]

Existem algumas macros que definem valores padrões de prioridades para a utilização em códigos escritos para programas do Linux, padronizando assim os valores das prioridades dos processos. Estas macros são fornecidas pelo *kernel* do Linux e podem ser vistas na figura 2.2. As macros *MAX_USER_RT_PRIO* e *MAX_RT_PRIO* definem o valor máximo

da prioridade de um processo de tempo real, enquanto que a macro *MAX_PRIO* define o valor máximo da prioridade de um processo normal e a macro *DEFAULT_PRIO* define um valor que é utilizado como padrão para os processos normais criados no sistema. Para realizar alterações nas prioridades dos processos, é aconselhável utilizar-se das macros como base para os valores máximos possíveis das prioridades, como também é necessário atenção para não inverter os valores reais que se quer como prioridade de um processo.

```
<linux/sched.h>

#define MAX_USER_RT_PRIO        100
#define MAX_RT_PRIO             MAX_USER_RT_PRIO

#define MAX_PRIO                (MAX_RT_PRIO + 40)
#define DEFAULT_PRIO            (MAX_RT_PRIO + 20)
```

Figura 2.2: Macros de prioridades de processos.

2.2 Ciclo de Vida dos Processos

Todo processo em um sistema operacional possui um ciclo de vida, o qual tem estados e regras para a transição entre estes estados. Cada processo pode estar apenas em um estado por vez e só podem mudar para outro de acordo com as regras de transição.

Segundo [5] e [31], um processo que esteja ativo no sistema pode estar em um dos seguintes estados:

- Executando (*running*): o processo está executando no momento.
- Esperando (*waiting*): o processo está apto a executar, mas por algum motivo o processador está sendo utilizado por outro processo. Neste caso, na próxima troca de processos, o escalonador pode selecionar este processo para utilizar o processador.
- Dormindo (*sleeping*): o processo está dormindo e não pode executar porque ele está esperando por um evento externo. Neste caso, na próxima troca de processo, o escalonador não pode selecioná-lo, já que ele espera por um evento e de nada adianta voltar ao processador sem que termine esta espera.

Para gerenciar os processos do sistema, são utilizados dois tipos de filas, a fila de prontos e a fila de espera do processador. Na fila de prontos ficam todos os processos que estão aptos a executar, como também o que está executando no processador, enquanto que na fila de espera estão todos os processos que estejam dormindo. A partir do momento que um processo que esteja na fila de espera seja acordado pelo evento que espera, ele volta para a fila de prontos do processador.

Quando um processo é criado, ele geralmente inicia na fila de prontos do processador e no estado "esperando", pois ele ainda deve ser analisado pelo escalonador, o qual decidirá quando cada processo deve ser executado. Analisando a figura 2.3 e considerando que o processo iniciou no estado "esperando", ou seja, ele está apto para executar, mas por algum motivo ainda não está executando, quando o processador decidir que o processo deve ser executado, ele passa pela transição 1 e muda para o estado "executando". A partir de então, ele pode ser escalonado novamente, voltando para o estado "esperando"(transição 2), pode esperar por algum evento externo, mudando para o estado "dormindo"(transição 3), ou ainda continuar sua execução e quando não tiver mais nada a ser processado em nenhum momento posterior, passa para o estado "parado"através da transição 5. Quando o processo está no estado "dormindo"e um evento externo o acorda, ele só pode passar pela transição 4, indo para o estado "esperando", para assim poder ser escalonado novamente. O estado parado é um estado em que o processo não executa mais, podendo ser considerado o estado final do processo.

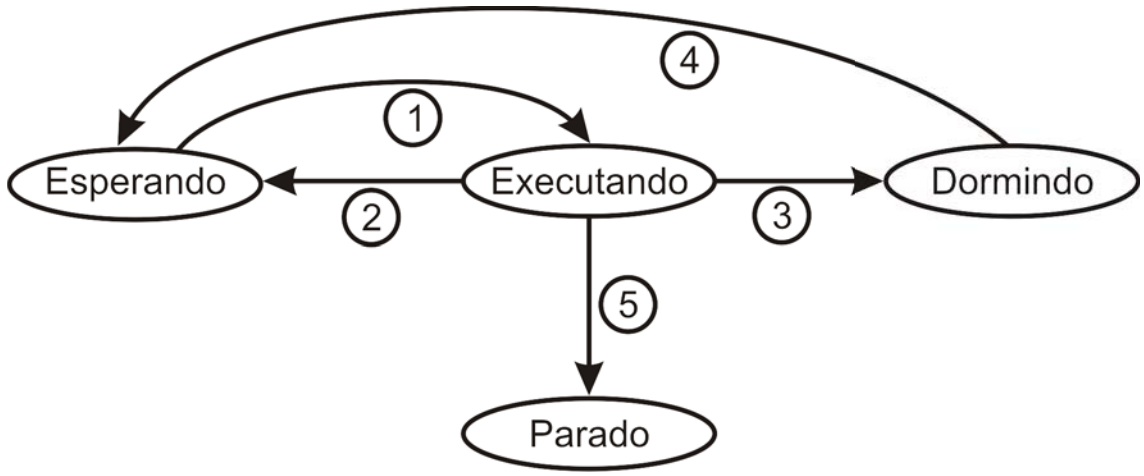

Figura 2.3: Ciclo de vida de um processo no Linux.

Além destes possíveis estados representados na figura 2.3, no Linux existe um estado chamado zumbi (*zombie*). Um processo entra neste estado quando todos os recursos alocados para ele foram liberados, ou seja, ele não pode mais executar nada, mas seu registro no sistema não foi desvinculado, ficando de certa forma ativo para o sistema, mas não podendo mais executar. Todos os processos quando vão ser finalizados passam por este estado, pois primeiramente eles têm seus recursos liberados, só depois o sistema é avisado de que aquele processo pode ser desvinculado. As vezes um processo cria outro processo, só que na hora de avisar sobre a finalização deste outro, o processo pai pode ter sido mal programado e não realiza da forma correta o aviso para o sistema desvincular ele, deixando-o em um estado "zumbi"até a reinicialização do sistema. Baseando-se na figura 2.3, este estado estaria em algum lugar entre o estado "executando"e o estado "parado", passando pela transição 5.

2.3 *Threads* de *Kernel*

Threads de *kernel* são processos criados diretamente pelo próprio *kernel*. Geralmente são criados quando o *kernel* precisa delegar funções para um processo separado, o qual é executado em "paralelo". Estas *threads* são usadas geralmente para [19] [31]:

- Sincronizar periodicamente páginas de memória modificadas.
- Escrever páginas de memórias na área de *swap* se elas são raramente utilizadas.
- Realizar postergação de trabalho.

Existem dois tipos de *threads* de *kernel* basicamente [31]:

- Tipo 1: A *thread* é iniciada e espera até que uma ação específica seja requisitada pelo *kernel*.
- Tipo 2: Uma vez iniciada, a *thread* executa em intervalos periódicos, verifica a utilização de um recurso específico e realiza alguma ação. Este tipo de *thread* é geralmente utilizado para o monitoramento contínuo de processos.

Como estas *threads* são criadas pelo próprio *kernel*, eles não executam em modo de usuário, mas em um modo supervisor, ou seja, no modo *kernel*, podendo acessar a parte do espaço de endereço do *kernel*.

Existem funções do *kernel* para criar estas *threads* e gerenciá-las. A função mais utilizada para criar uma *thread* de *kernel* é a *kthread_create*, a qual tem sua assinatura mostrada na figura 2.4. Esta função recebe como parâmetros um ponteiro para uma função, que é a qual será executada pela *thread*, um ponteiro para algum tipo de estrutura, ou algum dado que será passado para a função da *thread* e como terceiro parâmetro o nome da *thread* a ser criada.

```
<kernel/kthread.c>

struct task_struct *kthread_create(int (*threadfn)(void *data),
                                   void *data,
                                   const char namefmt[],
                                         ...)

void kthread_bind(struct task_struct *k, unsigned int cpu)
```

Figura 2.4: Função *kthread_create* e *kthread_bind*.

Uma função bastante utilizada em relação as *threads* de *kernel* é a *kthread_bind*, a qual é utilizada para ligar uma *thread* a uma determinada CPU, não permitindo que ela

seja executada em outra CPU do sistema. A assinatura da função *kthread_bind* pode ser visualizada na figura 2.4.

Depois que uma *thread* de *kernel* é criada através de *kthread_create*, ela ainda não está em execução, ela está na fila de *waiting* e precisa receber um evento para acordar. O evento é enviado para a *thread* através da função *wake_up_process*, fazendo com que a *thread* mude para a fila de prontos.

2.4 Interrupções

O Linux usa interrupções para realizar alguns processos necessários ao sistema. As interrupções utilizadas podem ser definidas como interrupções de *hardware* (*Hard IRQ*) ou interrupções de *software* (*SoftIRQ*). *Hard IRQs* são interrupções causadas pelo *hardware* com a intenção de chamar a atenção do processador para um determinado evento, o qual requer prioridade em seu processamento. As interrupções de *software* são geradas pelo sistema, objetivando postergar algum trabalho do próprio *kernel*.

Interrupções podem ser agrupadas em duas categorias (interrupções síncronas e exceções ou interrupções assíncronas). A primeira categoria é gerada pelo processador e são direcionadas ao processo em execução. As exceções podem ser geradas por causa de um erro de programação ocorrida em tempo de execução, como também por uma situação excepcional ou condição fora do previsto que pode ter acontecido e o processador precisa de ajuda para lidar com isto. As interrupções assíncronas são as interrupções clássicas geradas por dispositivos periféricos e pode ocorrer a qualquer momento [31].

O sistema do Linux pode executar em dois contextos diferentes, o contexto de interrupção e o de processo, onde o primeiro não pode acessar a memória virtual de usuário, já o segundo não pode acessar a memória do *kernel*. Sempre que uma interrupção for gerada e o sistema entra em contexto de interrupção e é executada uma rotina especial, a qual é denominada rotina de serviço de interrupção (*interrupt service routine - ISR*) também conhecida como tratador de interrupção (*interrupt handler*), responsável por tratar a interrupção e chamar a atenção do *kernel* para as alterações realizadas [19] [22].

Existem momentos que interrupções são desativadas no *kernel*, mas isso deve ser feito com cuidado, pois pode ocasionar o travamento ou grande atraso de outros processos do sistema [28]. Vale ressaltar que mesmo quando se desativa as interrupções, existem algumas que não podem ser desativadas por serem essenciais para a preservação do bom funcionamento do sistema.

Quando o sistema está executando uma rotina de serviço de interrupção uma ou mais interrupções são desativadas, então se a execução desta rotina for demorada, muitas interrupções podem ser perdidas, ocasionando problemas ao sistema. Para resolver este problema,

o tratador de interrupções foi dividido em duas partes. A primeira parte que é denominada de *top half* é constituída de todo o serviço que deve ser executado com urgência, necessitando de desempenho crítico e geralmente executa com uma ou mais interrupções desabilitadas. A segunda parte, denominada de *bottom half* é formada pelo serviço que não tem tanta urgência e pode ter sua execução postergada para um momento mais oportuno, tendo sua execução realizada com as interrupções habilitadas. A parte de execução denominada *top half* é executada pela interrupção de *hardware*, enquanto o *bottom half* é executado através de *softIRQ* [8] [19] [43].

Geralmente a rotina de um tratador de interrupção pode ser dividida em três partes em relação a sua importância para a execução do sistema [5] [31]:

- Ações críticas: Precisam ser executadas imediatamente após uma interrupção, para poder manter a estabilidade e correta operação do sistema. Outras interrupções precisam ser desabilitadas quando tais ações são realizadas.
- Ações não críticas: Devem ser executadas tão rapidamente quanto possível, mas pode ser executado com interrupções habilitadas.
- Ações postergáveis: Não são tão importantes e não precisam ser executadas diretamente pelo tratador de interrupção. O *kernel* pode postergar estas ações e processar elas quando não existir nada melhor para fazer.

As ações críticas e não críticas são processadas em *top half*, enquanto que as ações postergáveis são executadas em *bottom half*, através de *threads* de *kernel*. *Top half* pode ser referido também como *hard IRQ*, enquanto que *bottom half* pode ser referido por *softIRQ* [36].

2.5 A Classe de Escalonamento de Tempo Real

O Linux possui um escalonador global para realizar todo o escalonamento básico dos processos em seu sistema, mas existem processos que necessitam de regras diferentes para serem escalonados no sistema. Desta forma, Linux implementa uma técnica que permite esta diferenciação, chamada de classe de escalonamento. O Linux pode possuir várias classes de escalonamento, onde cada uma pode possuir suas regras de forma individual e se utilizar do escalonador genérico para realizar o trabalho em si. Assim o escalonador global pôde ser desenvolvido sem a necessidade do conhecimento das regras de cada tipo de processo no sistema [5] [31].

Entre as classes de escalonamento que podem existir no Linux, existem duas classes de escalonamento para os processos de tempo real, a classe *Round Robin* (*SCHED_RR*) e a

classe *First-in, first-out* (*SCHED_FIFO*), as quais podem ser definidas como a seguir [2] [4] [5]:

- *SCHED_FIFO*: Todos os processos escalonados por esta classe são divididos em filas, uma fila para cada prioridade diferente, assim como mostrado na figura 2.5. Quando um processo destes entra em execução, ele pode executar até o fim, ou pode ser preemptado por um processo de tempo real com prioridade maior que a dele. Uma vez terminando sua execução, o escalonador tira ele da fila de prontos na qual estava, mas caso ele precise executar novamente, ele volta para o final da fila de prontos, respeitando todos os processos de mesma prioridade que chegaram primeiro que ele na fila.

- *SCHED_RR*: Esta classe de escalonamento possui a mesma divisão de filas da classe *SCHED_FIFO*, mas com a diferença de que cada processo possui uma fatia de tempo para sua execução, a qual é reduzida a medida que ele é executado. Quando este valor chega a zero, ele é restaurado para o valor original, mas o processo pára sua execução e deixa que outro processo execute, indo para o final da sua fila de processos prontos. Desta forma, diferente da classe *SCHED_FIFO*, se um processo de tempo real quiser ficar executando por um tempo muito grande, ele não pára todos os outros processos, pois é escalonado de acordo com sua fatia de tempo.

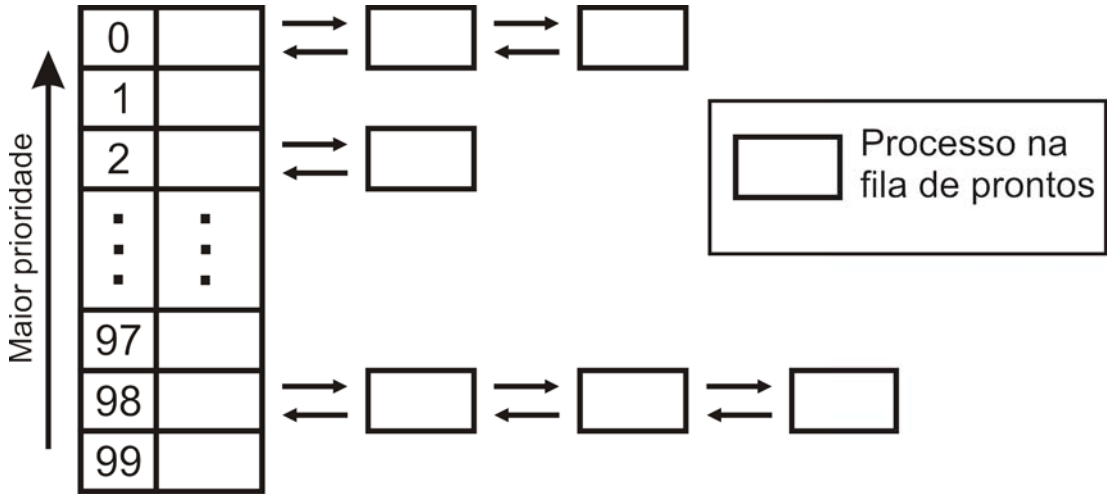

Figura 2.5: Fila de prontos do escalonador de tempo real.

Novas classes de escalonamento no Linux não podem ser inseridas dinamicamente, tendo que ser inserida em tempo de compilação do *kernel*. Quando qualquer classe for inserida no *kernel*, deve ser definido a sua prioridade em relação as outras classes. A classe de escalonamento de tempo real possui a mais alta prioridade no sistema, para assim poder gerenciar os processos de tempo real que devem ser executados com a maior precisão possível.

2.6 Considerações Finais

Este capítulo apresentou a teoria básica sobre processos e *threads* no Linux, como também interrupções e escalonamento de tempo real. Com estes conceitos pode-se obter um melhor entendimento do desenvolvimento deste trabalho, pois ele aborda a manipulação de *threads* e processos, assim como escalonamento de tempo real e manipulação de interrupções no Linux.

Capítulo 3

Temporizadores

Todo computador possui pelo menos um relógio em *hardware*, o qual conta a passagem do tempo e pode informá-la ao sistema. Os temporizadores em *software* utilizam-se destes relógios para realizar funções relacionadas ao tempo. Tais funções atualizam o tempo do sistema e informam aos processos que um determinado intervalo de tempo se passou.

A contagem de tempo é essencial para o funcionamento do sistema operacional. Ela informa qual o instante de um processo ser escalonado, por quanto tempo o processo deve utilizar o processador, por quanto tempo um processo periódico deve ficar em modo de espera, se um processo já esperou demais por outro processo ou não, entre outras utilidades.

Este capítulo trata de descrever quais relógios em *hardware* existem atualmente nos PCs, quais os tipos de temporizadores em *software* existentes e explicar suas implementações e funcionamento.

3.1 Relógios e Temporizadores em *Hardware*

O *kernel* precisa interagir com alguns tipos de relógios e temporizadores em *hardware*. Os relógios são utilizados para manter a hora do dia e realizar medições de tempo precisas. Os temporizadores em *hardware* são programados pelo *kernel*, de forma a gerar interrupções em uma frequência predefinida. A seguir é apresentada uma breve descrição sobre o relógio e os dispositivos de *hardware* que podem ser encontrados nos computadores [6].

3.1.1 Relógios de Tempo Real (RTC)

O relógio de tempo real (do inglês, *Real Time Clock* - RTC) é um relógio de *hardware* integrado ao CMOS RAM da placa mãe, geralmente implementado pelo circuito integrado

Motorola 146818. Ele está presente em todos os PCs atuais e sua função é registrar a data e hora atual. Ele é alimentado por uma pequena bateria, assim não para de funcionar nem quando o PC é desligado. Ele possui um erro de precisão esperado de 10 segundos a cada mês [34].

O RTC é capaz de enviar interrupções periódicas sobre a linha 8 das requisições de interrupção (do inglês, Interrupt Request - IRQ) em frequências entre 2 e 8.192 *hertz*, mas pode ser programado para ativar a IRQ 8 quando o RTC alcançar um valor específico, funcionando assim como um alarme [6].

O acesso de leitura ao RTC é lento, pois este é realizado através de uma sequência de instruções de entrada e saída (*inb* e *outb*), levando de 1 a 4 microssegundos para serem executadas a cada chamada. Estas instruções, além de lentas, ainda bloqueiam o processador enquanto estão sendo executadas, fazendo todo o sistema parar por alguns microssegundos. Devido a estas limitações, o RTC é utilizado nos sistemas operacionais apenas quando é necessário atualizar a hora do sistema ou quando o sistema operacional decide acertar a hora do RTC, a qual frequentemente pode ter desvios [1].

3.1.2 Contador de *Time Stamp* (TSC)

Todos os microprocessadores 80x86 possuem um pino de entrada CLK, o qual recebe um sinal de relógio de um oscilador externo. A partir do *Pentium*, os microprocessadores 80x86 possuem um contador que é incrementado a cada sinal do relógio. O contador é acessado através do registrador de 64 *bits* chamado Contador de *Time Stamp* (*Time Stamp Counter* - TSC), que pode ser lido através da instrução *rdtsc* em linguagem *assembly*. Quando se usa o TSC para medições de tempo deve-se levar em consideração a frequência de sinal do relógio, pois caso a frequência do relógio seja de 1 GHz, o TSC será incrementado a cada nanossegundo. O Linux pode utilizar-se dele para conseguir melhor precisão em medições de tempo do que utilizando o temporizador de intervalo programável ou PIT (*Programmable Interval Timer*)[6].

3.1.3 Temporizador de Intervalo Programável (PIT)

Computadores compatíveis com o PC IBM possuem outro tipo de dispositivo de medição de tempo, chamado Temporizador de Intervalo Programável (PIT). Cada computador compatível com o PC IBM possui pelo menos um PIT. Ele gera interrupções sempre em uma frequência fixa programada pelo kernel [6].

O Linux pode programar o PIT para enviar interrupções para a IRQ 0 em frequências diferentes dependendo da sua configuração. O kernel estudado estava configurado com o

valor padrão de 250 *hertz*, ou seja, uma interrupção a cada 4 milissegundos. Este intervalo é denominado de *tick* e o seu valor é guardado em nanossegundos na variável *tick_nsec*.

O PIT possui dois modos de operação: o modo *one-shot*, o qual gera uma interrupção única no momento desejado e o modo periódico, o qual gera interrupções periódicas. Os PCs modernos possuem todas as funções do PIT implementadas no *chipset* do PC [1].

3.1.4 Temporizador Local de CPU

O Controlador de Interrupção Programável Avançado (do inglês, Advanced Programmable Interrupt Controller - APIC) está presente em microprocessadores 80x86 recentes. Ele é incluído dentro da pastilha do processador e disponibiliza um temporizador local de CPU como mais um dispositivo de medição de tempo. Este temporizador é semelhante ao PIT, com algumas diferenças [1] [6]:

- O contador do temporizador do APIC é do tipo long de 32 *bits*, enquanto o do PIT é do tipo long de 16 *bits* e o temporizador local pode ser programado para usar interrupções em frequências muito baixas;
- O temporizador local APIC envia uma interrupção apenas para seu processador, enquanto o PIT causa uma interrupção global, que pode ser manipulada por qualquer CPU no sistema;
- O temporizador do APIC é baseado no sinal do barramento do relógio. Ele pode ser programado de forma a decrementar o contador do temporizador a cada 1, 2, 4, 8, 16, 32, 64 ou 128 sinais do barramento. De modo oposto, o PIT, que faz uso do seu próprio sinal de relógio, pode ser programado de forma mais flexível.

3.1.5 Temporizador de Eventos de Alta Precisão (HPET)

O HPET (High Precision Event Timer) é um novo *chip* de temporizador, desenvolvido juntamente pela Intel e Microsoft para a arquitetura PC. É suportado pelo Linux desde a versão 2.6 [6] [42]. Ele provê um número de temporizadores de *hardware* que podem ser explorados pelo *kernel*. Basicamente, o *chip* inclui contadores independentes de 32 ou 64 *bits*. Cada contador é orientado por um sinal de relógio, cuja frequência precisa estar em um mínimo de 10 MHz. Assim o contador é incrementado no mínimo uma vez em 100 nanossegundos. Cada contador destes é associado com no máximo 32 temporizadores e são compostos por um registrador *comparator* e um *match*. O *comparator* é um circuito que compara o valor do contador com o valor do registrador *match*, gerando uma interrupção de *hardware* se forem iguais. Alguns dos temporizadores também podem ser habilitados para gerar uma interrupção periódica [6] [27].

Os temporizadoresdo HPET são definidos de tal forma que, no futuro, o sistema operacional pode ser capaz de atribuir temporizadores específicos para ser utilizado diretamente por aplicações específicas [27]. O *chip* HPET pode ser programado através de endereços mapeados no espaço de memória. O BIOS estabelece o mapeamento durante a fase de inicialização do sistema e informa para o *kernel* do SO o endereço inicial de memória. O registrador HPET permite ao kernel ler e escrever os valores dos contadores e do registrador *match*, programar temporizadores *one-shot* e habilitar ou desabilitar interrupções periódicas dos temporizadores que suportam isto [6].

3.1.6 Temporizador Gerenciador de Energia ACPI (ACPI PMT)

O temporizador gerenciador de energia da *interface* de energia e configuração avançada (do inglês, Advanced Configuration and Power Interface Power Management Timer - ACPI PMT) é outro dispositivo de relógio incluído em quase todas as placas-mãe baseadas em ACPI. Seu sinal de relógio tem uma frequência fixa de aproximadamente 3,58 MHz. O dispositivo é um simples contador incrementado a cada *tick* de relógio. Para a leitura do valor atual do contador, o *kernel* acessa uma porta de E/S cujo endereço é determinado pela fase de inicialização do BIOS [6].

O ACPI PMT é preferível ao TSC se o sistema operacional ou o BIOS puderem diminuir dinamicamente a frequência ou a voltagem da CPU para economizar a energia da bateria. Quando isto acontece, a frequência do TSC muda, causando falha temporal e outros efeitos desagradáveis enquanto a frequência do ACPI PMT continua a mesma. Por outro lado, a alta frequência do contador do TSC é útil para medições muito pequenas de intervalo de tempo. Contudo se um dispositivo HPET estiver presente, ele sempre será preferível que outros circuitos, devido a sua alta precisão [6].

3.2 Visão Geral de Temporizadores em Software no Linux

Ao longo dos anos foram realizados vários estudos sobre temporizadores e maneiras de aperfeiçoá-los [39] [41] [10] [11] [12]. Primeiro surgiu a idéia de temporizadores baseados em uma frequência pré-definida, realizando-se tarefas do sistema operacional a cada *tick*. Desta forma é gerada uma interrupção de temporizador a cada *tick*, ativando o tratador de interrupções do temporizador, o qual verifica e executa as operações necessárias vinculadas aos temporizadores que estão expirando naquele momento.

Durante o início do desenvolvimento dos temporizadores, foi difícil realizar novos trabalhos e idéias para aprimorá-los. Um dos grandes problemas nos trabalhos e implementações sugeridos foi a dependência de hardware, pois causava grande complexidade e altos custos

de manutenção. Era necessário fazer uma implementação do trabalho proposto para cada arquitetura de computador diferente. Não existia nenhuma camada de abstração ou um subsistema genérico para facilitar o desenvolvimento de tais trabalhos [24].

Para facilitar o desenvolvimento dos temporizadores, foi desenvolvido um modelo de subsistema de tempo genérico com mínima dependência de arquitetura [14] [24]. Este modelo alterou de forma significante o subsistema de tempo do kernel do Linux, inserindo um modelo genérico para o registro e utilização das fontes e eventos de relógio, alterando também a forma de representar o tempo do sistema.

Com a evolução do hardware, foram constatadas novas necessidades relacionadas aos temporizadores, como por exemplo, economia de energia. O processador necessita a cada *tick* verificar se existem temporizadores a serem executados, caso existam, as funções vinculadas a elas devem ser executadas. Desta forma o processador nunca consegue ficar inativo por muito tempo e isto torna-se um problema para sistemas que necessitam economizar energia, pois mesmo que o processador esteja ocioso, ele deve realizar pelo menos uma verificação a cada *tick*. Desta forma, é utilizado processamento desnecessário quando não houver temporizadores a expirarem no momento, não permitindo que o processador entre em estado de economia de energia por longos períodos de tempo.

Tentando resolver a necessidade de economia de energia e ainda acrescentando a idéia de temporizadores com maior precisão temporal, surgiu a idéia de utilizar *ticks* dinâmicos no lugar de *ticks* periódicos. Desta forma, quando o processador estiver ocioso e não existir temporizadores a expirar em breve, o sistema pode entrar por períodos mais longos de economia de energia [10] [14].

Na tentativa de melhorar os temporizadores e sua precisão, foram desenvolvidos os temporizadores de alta resolução. Tais temporizadores se destacam dos anteriores, pois se utilizam dos *ticks* dinâmicos para causar interrupções apenas no momento em que os temporizadores de alta resolução expiram, também oferecendo precisão de nanossegundos para os temporizadores.

3.2.1 Tempo Global do Sistema

O tempo do sistema ou *time of day* (TOD) representa o tempo atual do sistema em nanossegundos. Ele precisa incrementar o *clock* do sistema monotonicamente, atualizar de forma precisa e rápida o tempo do sistema e fazer ajustes quando necessário.

Nas implementações existentes na maioria das arquiteturas, o TOD era atualizado a cada *tick* do sistema. Isso o tornava menos preciso, pois ele seria atualizado a cada 4 ms quando HZ=250. Para melhorar a precisão, o TOD passou a utilizar uma fonte de tempo de alta resolução para auxiliar a fonte de *tick* e obter uma melhor granularidade de tempo.

Desta forma, quando é necessário obter o TOD, o valor retornado não terá apenas a precisão anterior de 4 ms, mas ele terá também o tempo passado desde o último *tick* até o momento de solicitação do TOD [39].

Essas formas de obter o TOD tendem ao erro, pois se a função de interpolação não cobrir o intervalo de *tick* inteiro, o tempo será adiantado indevidamente. Se em qualquer momento, o valor da função de interpolação for maior que o intervalo de *tick*, o tempo pode sofrer atrasos. Essas variações no TOD podem ocorrer devido a vários fatores tais como erros de calibração, alterações no valor de ajuste do tempo, atraso no tratador de interrupção, alterações na frequência da fonte de tempo e perda de *ticks* [39].

Devido a estes fatores de erro, foi proposta e implementada uma forma de resolver tais problemas, visando a exatidão do tempo no sistema. Como consequência foi retirada a idéia de interpolação. Sendo assim o tempo do sistema sempre é calculado da mesma forma, seja ele para atualizar o relógio ou para ser utilizado de alguma forma entre dois *ticks*. Desta maneira, não é mais necessário atualizar o TOD a cada *tick* do sistema, pois sempre que ele for atualizado, é calculado o intervalo de tempo em nanossegundos desde a última vez que ele foi atualizado. Dessa forma ele não precisa mais ser chamado a cada interrupção do temporizador.

Nas alterações incluem-se melhorias nos ajustes de tempo realizados através da internet e seus servidores de tempo, que é realizado de forma mais direta e consistente, evitando erros que ocorriam com o código anterior. Nas melhorias realizadas, um dos seus benefícios foi o compartilhamento do algoritmo entre todas as arquiteturas. Isso reduziu um grande número de código específico para cada arquitetura, diminuindo as redundâncias e simplificando a manutenção [39].

3.2.2 Fonte de Relógio

Todo computador possui uma ou mais fontes de relógio, sendo elas mais ou menos potentes, fornecendo maior ou menor precisão temporal. Cada fonte destas necessita de um código específico para ser acessada, então foram realizados alguns trabalhos sobre o código fonte do Linux, desenvolvendo assim uma camada de abstração a todas as fontes de relógio. Tal camada facilitou o desenvolvimento e aprimoramento do subsistema de tempo. Assim criou-se uma generalização das fontes de relógio, diminuindo a dependência de hardware existente anteriormente. Antes do seu desenvolvimento, para cada fonte de relógio e praticamente para cada arquitetura existia uma implementação. Desta forma existia muito código duplicado desnecessariamente [24].

O código para gerenciar as fontes de relógio possui estruturas utilizadas para definir os relógios no sistema e uma interface de programação de aplicativos com funções para ler

e converter valores de ciclo do relógio para nanossegundos, efetuar o registro de tais fontes, selecionar a melhor fonte e outras mais [39].

```
<linux/clocksource.h>

struct clocksource {

  char              *name;
  struct list_head  list;
  int               rating;
  cycle_t           (*read)(struct clocksource *cs);
  cycle_t           mask;
  u32               mult;
  u32               shift;
  unsigned long     flags;
...
};
```

Figura 3.1: Estrutura da fonte de relógio.

Na figura 3.1 pode ser verificada as variáveis mais importantes da estrutura de dados utilizada para gerenciar as fontes de relógio no *kernel*. O significado das variáveis da estrutura é definido como:

- *name*: Representa um nome legível da estrutura, como por exemplo, HPET, PIT, TSC.
- *list*: É o elemento de uma lista onde as fontes de relógio são registradas.
- *rating*: É utilizada como classificação da qualidade da fonte de relógio. Fontes com valores entre 1 e 99 são as piores, sendo utilizadas apenas na inicialização do sistema ou quando não houver nada melhor. Entre 100 e 199 são fontes adequadas para o uso, caso não tenha nenhuma fonte melhor. Entre 200 e 299 são fontes boas e usáveis. Entre 300 e 399 são razoavelmente rápidas e precisas. Fontes com *rating* entre 400 e 499 são consideradas ideais.
- *read*: É o ponteiro para uma função utilizada para ler o valor de ciclo atual do relógio.
- *mask*: A função do ponteiro *read* deve retornar um valor de 64 *bits*. Caso o relógio não disponha de valores de tempo com 64 *bits*, o campo *mask* é utilizado para especificar uma máscara de *bits* (*bitmask*) para selecionar os *bits* apropriados.
- *mult* e *shift*: Como o valor retornado pela função do ponteiro *read* não usa uma base de tempo fixa para todos os relógios, para efetuar a conversão para nanossegundos, utiliza-se *mult* para multiplicar e *shift* para dividir o valor de ciclo retornado.
- *flags*: Representa as *flags* do relógio, indicando algumas informações de status para cada relógio.

As fontes de relógio são registradas no sistema através das funções disponibilizadas pelo *kernel*. Elas são ordenadas pela sua taxa de qualidade em uma lista global. O sistema utiliza a fonte com melhor taxa de qualidade por padrão, mas isso pode ser alterado pelo usuário, podendo definir qualquer outra fonte.

3.2.3 Dispositivos de Eventos de Relógio

Fontes de eventos de relógio ou dispositivos de eventos de relógio representam os eventos que ocorrem no sistema em algum tempo no futuro. Eles são programados para avisar quando um evento ocorre e, então, tratá-lo.

A figura 3.2 apresenta os campos que compõem a estrutura utilizada para registrar e gerenciar os dispositivos de eventos de relógio, detalhadas a seguir:

```
<linux/clockchips.h>

struct clock_event_device {
  const char            *name;
  unsigned int          features;
  unsigned long         max_delta_ns;
  unsigned long         min_delta_ns;
  unsigned long         mult;
  int                   shift;
  int                   rating;
  int                   irq;
  const struct cpumask  *cpumask;
  int                   (*set_next_event)
      (unsigned long evt, struct clock_event_device *);
  void                  (*set_mode)
      (enum clock_event_mode mode, struct clock_event_device *);
  void                  (*event_handler) (struct clock_event_device *);
  void                  (*broadcast) (const struct cpumask *mask);
  struct list_head      list;
  enum clock_event_mode mode;
  ktime_t               next_event;
};
```

Figura 3.2: Estrutura dos dispositivos de eventos de relógio.

- *name*: é o nome do dispositivo em questão. No Linux, os nomes dos dispositivos podem ser listados através de */proc/timerlist*.
- *features*: identifica as características do dispositivo. Tais características podem identificar se o dispositivo suporta eventos periódicos, eventos que ocorrem apenas uma vez (*one-shot*) ou se o dispositivo pode ser desativado, como é o caso específico dos APICs locais, que, em determinados níveis de economia de energia, são desligados.
- *max_delta_ns* e *min_delta_ns*: representam a diferença máxima e mínima, respectivamente, entre o tempo atual e o tempo até o próximo evento.

- *mult* e *shift*: Cada relógio possui uma frequência de oscilação própria, então essas variáveis são utilizadas para transformar o tempo do relógio de ciclos para nanossegundos.

- *rating*: esta variável permite a comparação dos dispositivos de eventos de relógio pelo seu grau de precisão, podendo especificar qual o mais preciso.

- *irq*: especifica o número da IRQ usada pelo dispositivo de evento de relógio. Esse valor só é necessário quando o dispositivo é configurado como global em relação a todas as CPUs do sistema. Dispositivos locais usam mecanismos de *hardware* diferentes para emitir sinais. Neste caso, eles configuram essa variável como -1.

- *cpumask*: especifica para quais CPUs o dispositivo de eventos está associado. Os dispositivos locais são geralmente responsáveis por apenas uma CPU.

- *set_next_event*: é um ponteiro para uma função que configura o próximo evento. Mas códigos genéricos não precisam chamar esta função diretamente, pois o *kernel* provê uma função para auxiliar nesta tarefa (*clockevents_program_event*).

- *set_mode*: é um ponteiro para uma função que pode alternar o modo de operação entre o modo periódico e *one-shot*.

- *event_handler*: é um ponteiro para uma função que será chamada pelo código de *interface* do *hardware* (que geralmente é escrito para uma arquitetura específica), passando os eventos de relógio para a camada genérica.

- *broadcast*: é um ponteiro para uma função que implementa o modo de *broadcast*, o qual contorna o problema de APICs locais que não estejam em funcionamento devido a questões de economia de energia.

- *list*: todas as instâncias de dispositivos de eventos de *clock* são mantidas em uma lista global e essa variável aponta para o início dessa lista.

- *mode*: informa o modo atual de operação, podendo ser modo periódico ou *one-shot*.

- *next_event*: define o tempo absoluto em que o próximo evento ocorrerá.

Em sistemas IA-32 e AMD64 a variável *global_clock_event* (em *arch/x86/kernel/i8253*) define o dispositivo de eventos de relógio usado como dispositivo global. Nestes sistemas, o dispositivo de eventos de relógio global é definido inicialmente como PIT. Se no computador existir um HPET, ele é inicializado um pouco mais tarde que o PIT e é utilizado no lugar deste, alterando assim a variável *global_clock_event* [31].

Os dispositivos de relógio e os dispositivos de eventos de relógio são desconectados em nível de estrutura, ou seja, eles funcionam independente um do outro. No entanto, um *chip*

de *hardware* no sistema é capaz de satisfazer as requisições das duas *interfaces*. Assim, o *kernel* geralmente registra um dispositivo de relógio e um dispositivo de eventos de relógio por *chip* de *hardware* de tempo. Desta forma, são adicionados dois objetos gerenciadores de tempo para o *kernel*, mas apenas um dispositivo de *hardware* é utilizado [31].

3.2.4 Dispositivos de *Tick*

Um dispositivo de *tick* é uma extensão de um dispositivo de eventos de relógio usado para prover *ticks* periódicos. A estrutura deste dispositivo é mostrada na figura 3.3.

```
<linux/tick.h>

enum tick_device_mode {
  TICKDEV_MODE_PERIODIC,
  TICKDEV_MODE_ONESHOT,
};

struct tick_device {
  struct clock_event_device *evtdev;
  enum tick_device_mode mode;
};
```

Figura 3.3: Estrutura do dispositivo de *tick*.

A estrutura *tick_device* é uma extensão da estrutura *clock_event_device*, contendo apenas um campo a mais, o qual especifica o modo do dispositivo (periódico ou disparo único (*one-shot*)). Sempre que um dispositivo de evento de relógio é adicionado, o *kernel* automaticamente adiciona um dispositivo de *tick*.

O *kernel* distingue estes dispositivos entre aqueles de *tick* global ou local. Os dispositivos locais são organizados na lista definida por CPU. O dispositivo definido como global é informado por uma variável. As variáveis e funções que manipulam estas informações são definidas em *kernel/time/tick-internal.h*. As principais variáveis são definidas como:

- *tick_cpu_device*: é uma lista definida por CPU que informa os dispositivos de *tick* de cada CPU;
- *tick_next_period*: especifica o tempo (em nanossegundos) em que o próximo evento de *tick* global acontecerá;
- *tick_do_timer_cpu*: contém o número da CPU definida como dispositivo de *tick* global;
- *tick_period*: informa o valor do intervalo entre *ticks* em nanossegundos. Ele é a contraparte da variável HZ, que denota a frequência em que os *ticks* ocorrem.

Para configurar um dispositivo de *tick*, o *kernel* disponibiliza a função *tick_setup_device*. Nesta função, o dispositivo é configurado como dispositivo de *tick* global ou não, dependendo de qual dispositivo for escolhido para tal papel. O *kernel* verifica se o dispositivo está inativo devido a algum estado de economia de energia. Caso o dispositivo esteja ativo, a função estabelece um *tick* periódico. Esse *tick* é estabelecido de forma diferente, dependendo se o dispositivo está no modo periódico ou no modo de disparo único (*one-shot*).

3.3 Temporizadores de Baixa Resolução

Temporizadores de baixa resolução, conhecidos também como temporizadores clássicos, são utilizados pelo *kernel* do Linux desde seu início. Eles trabalham em um intervalo de tempo pré-definido, o que acaba limitando sua precisão temporal.

Eles são utilizados, tanto por funções do *kernel* do sistema operacional como por funções em espaço de usuário. Sua estrutura e funções sofreram diversas modificações com o passar do tempo, tentando assim melhorar sua resolução temporal e diminuir o seu custo de execução (*overhead*).

3.3.1 Frequência e Contagem do Tempo

O *kernel* do sistema operacional utiliza interrupções temporizadas de *hardware* geradas por um processador ou outra fonte periódica disponível como base de tempo para os temporizadores do sistema. A frequência de interrupções do sistema também conhecida como taxa de *tick* é programada durante a sua inicialização. O valor desta taxa é definido em uma variável pré-definida chamada HZ. O valor de HZ é diferente para cada arquitetura suportada e é definido no *kernel* do Linux no arquivo *asm-arch/param.h*. A taxa de *tick* tem uma frequência de HZ *hertz* e um período de 1/HZ por segundo. Então, por exemplo, se HZ for igual a 1000 *hertz*, ocorrerá uma interrupção a cada milissegundo, ou seja, ocorrem 1000 interrupções por segundo [30].

O valor de HZ pode fazer uma grande diferença no desempenho do sistema operacional. Quanto maior for o seu valor, mais precisão temporal o *kernel* garante. Contudo, isso aumenta a sobrecarga do sistema, pois as interrupções são geradas em um período menor de tempo. Por exemplo, se HZ for igual a 100 a granularidade dos temporizadores será de 10 milissegundos, ou seja, todos os eventos periódicos que se utilizam dos temporizadores de baixa resolução serão executados em intervalos múltiplos de 10 milissegundos. No entanto, quando se altera o HZ para 1000, a granularidade do sistema passa de 10 para 1 milissegundo, permitindo que os processos sejam executados com uma precisão maior de tempo. Atualmente o padrão para HZ no Linux é de 250 *hertz*, ou seja, ocorre uma interrupção a cada 4 milissegundos.

Uma melhor resolução e precisão de temporizadores de baixa resolução acrescentam algumas vantagens ao sistema, tais como:

- Chamadas de sistema que utilizam os temporizadores de baixa resolução executam com melhor precisão;
- Preempções de processos ocorrem de forma mais precisa, melhorando o escalonamento de processos;
- Medições e atualizações realizadas dentro do sistema são geradas com uma resolução mais refinada.

Uma variável também muito importante na utilização de temporizadores de baixa resolução é a variável global *jiffies* declarada em *linux/jiffies.h*. Esta variável mantém o número de *ticks* que ocorreram desde que o sistema foi iniciado. Durante a inicialização do sistema, esta variável deveria ser inicializada com zero, mas ela é inicializada com $HZ*(-300)$ e depois a cada interrupção do temporizador ela é incrementada por um. Assim, ela será incrementada HZ vezes por segundo.

A variável jiffies sempre foi do tipo *unsigned long*, possuindo então o tamanho de 32 *bits* na arquitetura de 32 *bits* e 64 *bits* na arquitetura de 64 *bits*. Em uma arquitetura de 32 *bits*, com HZ igual a 100 *hertz*, se *jiffies* fosse inicializada em zero, demoraria cerca de 497 dias para ocorrer um *overflow* na variável *jiffies*. Como atualmente HZ é igual a 250 *hertz* no Linux, demoraria cerca de 198 dias para ocorrer este *overflow* e quando o HZ for igual a 1000 *hertz* bastam apenas cerca de 49,7 dias para ocorrer este problema. Mas como *jiffies* é inicializado com outro valor, esse *overflow* ocorrerá em apenas 5 minutos, independente de qual seja o valor de HZ. Dessa forma, falhas no código do kernel relacionados com o *overflow* de *jiffies* surgirão logo na fase de desenvolvimento e não passarão despercebidas para versões estáveis do *kernel* [6].

Para resolver o problema de *overflow* da variável *jiffies* de 32 bits, foi criada uma variável *jiffies_64* que possui um tamanho de 64 *bits*. Assim o *overflow* sobre esta variável, quando HZ for igual a 1000, só ocorreria em algumas centenas de milhões de anos, podendo-se considerar então que não ocorrerá este *overflow*.

Mesmo depois da criação da variável *jiffies_64*, manteve-se a variável *jiffies* de 32 *bits* devido a compatibilidade com código *kernel* existente e devido ao desempenho, pois é bem mais simples e rápido acessar uma variável de 32 *bits* em uma arquitetura de 32 *bits* em vez de uma variável com o dobro do tamanho. Sendo assim, a variável *jiffies* acessa os 32 *bits* menos significativos da variável *jiffies_64* [30]. A figura 3.4 ilustra o esquema de acesso entre as variáveis *jiffies* e *jiffies_64*.

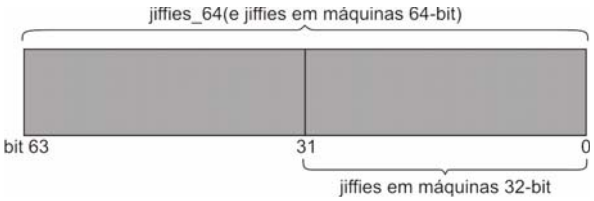

Figura 3.4: Layout de *jiffies* e *jiffies_64* [6].

Quando se acessa diretamente a variável *jiffies*, são obtidos apenas os 32 *bits* menos significativos de *jiffies_64*. Para acessar o valor de 64 *bits*, o *kernel* disponibiliza a função *get_jiffies_64()*. Esta função tenta acessar *jiffies_64* até ter certeza que ela não foi atualizada concorrentemente por outra parte do código do *kernel*. Esta verificação é realizada através de uma variável de bloqueio *xtime_lock* do tipo *seqlock*, a qual é utilizada na atualização e leitura de *jiffies_64* [6].

O *kernel* do Linux define algumas macros para comparação de *ticks* corretamente. É recomendada a utilização delas quando se trabalha com *jiffies*, as quais são declaradas em *linux/jiffies.h* e podem ser consultadas no apêndice A.

3.3.2 Estruturas de Dados dos Temporizadores de Baixa Resolução

O *kernel* do Linux representa um temporizador de baixa resolução como sendo uma variável do tipo *timer_list*. Este tipo é definido como uma estrutura de dados com algumas variáveis necessárias ao funcionamento do temporizador, como pode ser verificada na figura 3.5.

```
<linux/timer.h>

struct timer_list {
  struct list_head entry;
  unsigned long expires;

  void (*function)(unsigned long);
  unsigned long data;

  struct tvec_base *base;
...
};
```

Figura 3.5: Variáveis mais significativas da estrutura do *timer_list*.

A seguir é apresentada a definição das variáveis da estrutura *timer_list* apresentadas na figura 3.5, alguns outros campos desta estrutura são de uso auxiliar e não são discutidos neste trabalho:

- *entry*: Como os temporizadores são ligados entre si através de uma lista duplamente

encadeada, esta variável representa a cabeça da lista, sendo configurada assim sempre que o temporizador for inserido na fila.

- *expires*: Especifica o tempo absoluto em que esse temporizador expira. O tempo é informado em *jiffies*.
- *function*: É um ponteiro para uma função que deve ser chamada quando o temporizador expira.
- *data*: É um argumento passado para a função quando ela for ser executada.
- *base*: É um ponteiro para um elemento base onde os temporizadores são ordenados pelo tempo em que eles expiram. Existe um elemento destes para cada processador no sistema.

O *kernel* utiliza valores de tempo absoluto e relativo. Quando se inicia um temporizador, é comum utilizar valores relativos de tempo, indicando que o temporizador deve expirar em um tempo qualquer a partir do momento de sua criação. Valores absolutos são utilizados pelas estruturas de dados, pois assim facilita a comparação com o valor de *jiffies*, já que este também é guardado como valor absoluto.

Para facilitar o trabalho dos programadores no momento de definir um valor de tempo qualquer, foram criadas duas estruturas de dados que podem especificar o tempo em segundos e microssegundos ou nanossegundos em vez de definir esse tempo em quantidade de *ticks*. Junto com essas estruturas são declaradas funções para convertê-las de suas respectivas unidades de tempo para a unidade de tempo utilizada por *jiffies* e vice-versa. Essas estruturas e as assinatureas destas funções podem ser verificadas na figura 3.6.

```
<linux/time.h>

struct timeval {
  time_t       tv_sec;  /* segundos */
  suseconds_t  tv_usec; /* microssegundos */
};

struct timespec {
  time_t       tv_sec;  /* segundos */
  long         tv_nsec; /* nanossegundos */
};

unsigned long timeval_to_jiffies(const struct timeval *value);
void jiffies_to_timeval(const unsigned long jiffies, struct timeval *value);

unsigned long timespec_to_jiffies(const struct timespec *value);
void jiffies_to_timespec(const unsigned long jiffies,
                         struct timespec *value);
```

Figura 3.6: Estruturas *timeval* e *timespec* e funções de conversão.

Um ponto muito importante para o desempenho dos temporizadores é a forma como eles são organizados. Os temporizadores devem ser armazenados de uma forma que possam ser inseridos, acessados e removidos rapidamente. Antigamente os temporizadores clássicos ou de baixa resolução eram armazenados em listas duplamente encadeadas, mas esse tipo de estrutura era muito lento para inserção e pesquisa. Em 1997, foi implementada uma nova abordagem para a ordenação e acesso rápido aos temporizadores registrados no sistema, denominada *Cascading Time Wheel* (CTW) [24].

O CTW é formado por uma estrutura de dados declarada em *kernel/timer.c*. Ele é baseado no calendário de filas, que permite inserção, acesso e remoção ao temporizador em tempo O(1) [7] [20]. Mais informações sobre o calendário de filas podem ser encontradas no artigo de Brown em [7]. Em sua estrutura são definidas as variáveis necessárias para criar uma base de armazenamento dos temporizadores por CPU. Esta estrutura e suas principais variáveis podem ser verificadas na figura 3.7.

```
<kernel/timer.c>

#define TVN_BITS (CONFIG_BASE_SMALL ? 4 : 6)
#define TVR_BITS (CONFIG_BASE_SMALL ? 6 : 8)
#define TVN_SIZE (1 << TVN_BITS)
#define TVR_SIZE (1 << TVR_BITS)
#define TVN_MASK (TVN_SIZE - 1)
#define TVR_MASK (TVR_SIZE - 1)

struct tvec {
  struct list_head vec[TVN_SIZE];
};

struct tvec_root {
  struct list_head vec[TVR_SIZE];
};

struct tvec_base {
...
  struct timer_list *running_timer;
  unsigned long timer_jiffies;
  struct tvec_root tv1;
  struct tvec tv2;
  struct tvec tv3;
  struct tvec tv4;
  struct tvec tv5;
} ____cacheline_aligned;
```

Figura 3.7: Estrutura de dados do CTW.

A base para gerenciar os temporizadores de baixa resolução é a *tvec_base*, a qual é composta por outras variáveis, onde as principais são descritas a seguir:

- *running_timer*: É um ponteiro que aponta para o temporizador que expirou e tem sua função em execução no momento.

- *timer_jiffies*: Registra o tempo em *jiffies* de todos os temporizadores expirados registrados na base e que já expiraram. Normalmente ele tem valor igual ou um a menos que o valor de *jiffies*, podendo variar se as interrupções do processador estiveram momentaneamente desabilitado em relação a execução de temporizadores de baixa resolução [31].

- *tv1*: É um vetor de listas duplamente encadeadas de tamanho igual a 256 ou 64, dependendo da opção *CONFIG_BASE_SMALL*. Esta opção foi inserida para o caso da disponibilidade de pouca memória na máquina, como é o caso geralmente de sistemas embarcados. Este vetor é utilizado para ordenar os temporizadores que estão para expirar entre os próximos 255 *jiffies*, considerando o vetor de tamanho 256.

- *tv2*, *tv3*, *tv4* e *tv5*: São 4 vetores de listas duplamente encadeadas de tamanho igual a 64 ou 16, dependendo da opção *CONFIG_BASE_SMALL*. Estes vetores diferentemente do *tv1*, dividem os temporizadores em faixas pelos tempos em que eles expiram, cada posição representa um intervalo de expirações e cada variável representa um tamanho de intervalo diferente.

A figura 3.8 apresenta uma ilustração de como são organizados os temporizadores na estrutura de dados *tvec_base*.

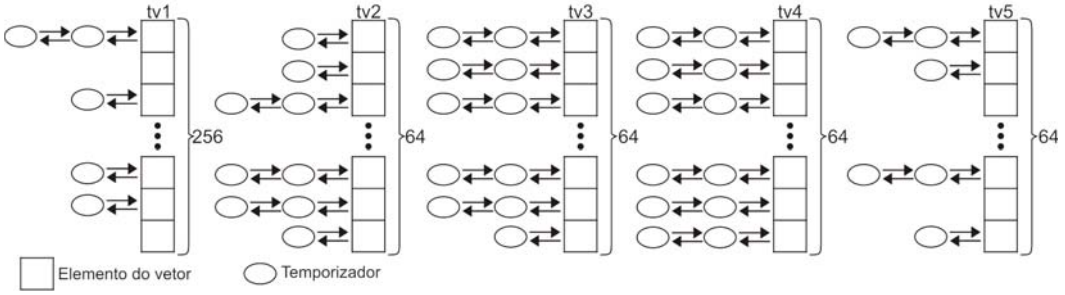

Figura 3.8: Representação da estrutura de dados do CTW.

Como visto nas figuras 3.7 e 3.8, existem 5 vetores para ordenar os temporizadores de baixa resolução. Esta ordenação divide os temporizadores em grupos pelo tempo em que eles expiram. Considerando sempre aqui a opção *CONFIG_BASE_SMALL* como falsa, o vetor *tv1* possui 256 posições e ele armazena todos os temporizadores que expirarão entre os próximos 255 *jiffies*. Os outros quatro vetores possuem 64 posições cada um, representando faixas de tempo diferentes como demonstrado na tabela 3.1.

Cada posição do vetor *tv1* representa um único valor de tempo em *jiffies*, no qual pode existir vários temporizadores de baixa resolução com o mesmo tempo em que devem expirar, ou seja, vários temporizadores que expirarão ao mesmo tempo(enfileirados em uma estrutura FIFO (*First In First Out*)). Cada posição do vetor *tv2* representa uma faixa de 256 *ticks*. Desta forma, cada posição de *tv2* representa uma faixa equivalente a faixa que todo o vetor

Vetor	Intervalo em *jiffies*
tv1	0 ... 255
tv2	256 ... 16.383
tv3	16.384 ... 1.048.575
tv4	1.048.576 ... 67.108.863
tv5	67.108.864 ... 4.294.967.295

Tabela 3.1: Intervalos dos vetores da estrutura CTW.

tv1 representa. Assim, cada posição destas é suficiente para preencher todas as posições do vetor *tv1*. Todos os temporizadores em *tv2* são enfileirados por listas duplamente encadeadas, nas quais cada posição no vetor aponta para uma lista diferente, assim como ocorre em *tv1* e nos outros três vetores.

O vetor *tv3* possui 64 posições, cada uma representando faixas de 16.384 *ticks* (64 * 256), sendo capaz de cada posição preencher todas as posições de *tv2* quando necessário. Seguindo o mesmo raciocínio, cada uma das 64 posições do vetor *tv4* possuem faixas de 1.048.576 *ticks* (64 * 64 * 256). Por sua vez o vetor *tv5* que possui a maior abrangência de valores de 67.108.864 *ticks* (64 * 64 * 64 * 256) para cada posição do vetor.

3.3.3 Utilização e Funcionamento dos Temporizadores de Baixa Resolução no *Kernel*

Os temporizadores de baixa resolução são representados dentro do *kernel* pela estrutura *timer_list*. A utilização deles é muito simples, tratando-se de definir uma variável deste tipo, a qual é necessária para representar o temporizador. Portanto, pode-se utilizar a macro *DEFINE_TIMER* (figura 3.9), a qual define um temporizador de forma estática, vincula a função a ser executada pelo temporizador, o valor do campo data e o tempo que ele deve expirar. Outra opção consiste em criar a variável *timer_list* no próprio código e definir seu nome, função, a variável que deve ser passado para a função e o tempo em que ele deve expirar, tudo isso acessando diretamente as variáveis e atribuindo seus valores. Para inicializar o temporizador deve-se utilizar funções disponibilizadas pelo *kernel*.

```
<linux/timer.h>

#define DEFINE_TIMER(_name, _function, _expires, _data)
```

Figura 3.9: Definição de *DEFINE_TIMER*.

Na figura 3.10, são mostradas as assinaturas das funções disponibilizadas para a inicialização de um temporizador de baixa resolução. Todas essas funções são disponibilizadas para acesso pelo código do *kernel* ou via módulo do *kernel*, ou seja, programas em espaço de usuário não podem acessá-las, a menos que elas sejam exportadas como chamadas de sistema.

Todas as funções apresentadas na figura 3.10 inicializam um temporizador de baixa resolução. Elas recebem como parâmetros um ponteiro para o temporizador que deve ser inicializado, um ponteiro para o nome do temporizador e um ponteiro do tipo *lock_class_key*. Os dois últimos parâmetros são utilizados para registrar *locks* no sistema, registro que é utilizado na tentativa de evitar *deadlocks* no sistema operacional. Cada função dessas tem uma diferença em sua inicialização, sendo a *init_timer_key* a inicialização padrão, a função *init_timer_on_stack_key* inicializa um temporizador e passa informações adicionais ao sistema por funções de *debug*.

A função *init_timer_deferrable_key* inicia um temporizador, mas configura-o através de uma *flag* como sendo um temporizador menos importante, podendo assim ter sua execução postergada. Desta forma, sempre que o sistema estiver sem processos para executar e puder dormir até que o próximo temporizador precise ser executado, todos os temporizadores com a *flag deferrable* habilitada serão desconsiderados. Assim, o sistema não precisa acordar só para executar estes temporizadores que podem ser adiados [15].

```
<kernel/timer.c>

void init_timer_key(struct timer_list *timer, const char *name,
                    struct lock_class_key *key)
{ ... }

void init_timer_on_stack_key(struct timer_list *timer,const char *name,
                             struct lock_class_key *key)
{ ... }

void init_timer_deferrable_key(struct timer_list *timer, const char *name,
                               struct lock_class_key *key)
{ ... }
```

Figura 3.10: Funções para inicializar um temporizador de baixa resolução.

Após a variável *timer_list* estar declarada e com seus campos devidamente inicializados, é necessário a inserção dela na fila de temporizadores, pois o temporizador não está em execução até que ele seja inserido na fila gerenciada pelo sistema. Para a inserção de temporizadores existem as funções *add_timer* e *add_timer_on* como mostrado na figura 3.11.

```
<kernel/timer.c>

void add_timer(struct timer_list *timer)
{ ... }

void add_timer_on(struct timer_list *timer, int cpu)
{ ... )
```

Figura 3.11: Funções utilizadas para inserção de temporizadores na lista.

A função *add_timer* recebe como parâmetro um ponteiro para o temporizador que deve estar devidamente inicializado e que será inserido na lista de temporizadores, passando a estar

ativo desta forma. Ela adiciona o temporizador na sua respectiva fila de acordo com o valor do seu campo *expires*. A função *add_timer_on* recebe um ponteiro para o temporizador a ser inserido e o número da CPU onde este deve ser inserido. A função realiza alguns procedimentos para averiguar se o temporizador ainda não foi inserido e faz a sua devida inserção na fila de temporizadores certa. Após a inserção o temporizador está ativo e pronto para executar assim que seu tempo expirar. Temporizadores podem ser reutilizados, modificando o tempo em que ele deve expirar. Para isso são disponibilizadas as funções *mod_timer*, *mod_timer_pending* e *mod_timer_pinned* como demonstrado na figura 3.12. Todas estas funções recebem como parâmetro um ponteiro para o temporizador a ser alterado e o novo valor de tempo em que ele deve expirar. Elas podem ser utilizadas para alterar tanto temporizadores ativos como os que já expiraram, sendo essas as formas mais seguras de alterar o tempo de um temporizador que ainda não expirou.

```
<kernel/timer.c>

int mod_timer(struct timer_list *timer, unsigned long expires)
{ ... }

int mod_timer_pending(struct timer_list *timer, unsigned long expires)
{ ... }

int mod_timer_pinned(struct timer_list *timer, unsigned long expires)
{ ... }
```

Figura 3.12: Funções para modificar um temporizador.

A função *mod_timer* faz as verificações necessárias para não realizar trabalho desnecessário alterando o temporizador para o mesmo valor de tempo que ele já tinha, realiza também os procedimentos de *debug* e se realmente necessário altera o valor de *expires* e ativa o temporizador. A função retorna 1 caso o temporizador estivesse ativo e 0 caso ele estivesse inativo no sistema.

A função *mod_timer_pending* altera o tempo em que o temporizador deveria expirar e ativa-o novamente, caso o temporizador estivesse ativo, caso contrário a função retorna 0 e não atualiza o temporizador nem o ativa. A função *mod_timer_pinned* altera o tempo em que o temporizador informado deveria expirar, não permitindo que ele migre para outro processador, logo depois ativa-o colocando na lista de temporizadores [3].

Muitas vezes é necessário desabilitar temporizadores, pois o trabalho que ele deveria realizar não é mais necessário. Um bom exemplo é em aplicativos de rede que enviam pacotes e tem que receber uma confirmação (ACK) do recebimento dele em um tempo predeterminado. Para contar este tempo é ativado um temporizador, o qual quando expira, avisa o aplicativo. Na maioria dos casos esta confirmação é realizada antes do tempo esgotar, assim o temporizador ativo não tem mais utilidade, sendo ele desativado. Para desativar temporizadores existem três funções, cujas assinaturas podem ser observadas na figura 3.13. A

função *del_timer* desativa o temporizador e retorna 1 caso ele estivesse ativo. Caso o temporizador já estivesse desativado a função retornaria 0 e não realizaria nenhum trabalho a mais. Garantindo assim que a função desse temporizador não será executada no futuro. Mas vale ressaltar que não é necessária a preocupação de desativar o temporizador depois que ele expirou, pois isso é feito automaticamente quando ele expira.

```
<kernel/timer.c>

int del_timer(struct timer_list *timer)
{ ... }

int del_timer_sync(struct timer_list *timer)
{ ... }

int try_to_del_timer_sync(struct timer_list *timer)
{ ... }
```

Figura 3.13: Funções para desativar temporizadores de baixa resolução.

A função *del_timer_sync* difere da função *del_timer* porque além de desabilitar o temporizador informado, ela prevê a possibilidade da função do temporizador estar sendo executada em alguma CPU. Caso isso esteja ocorrendo, ela espera até o fim dessa execução. Assim é garantido que a função atrelada ao temporizador não está sendo, nem será executada, depois que ele seja desativado. Por essa razão em quase todos os casos é aconselhável a utilização de *del_timer_sync* em vez de *del_timer* [9] [30].

A função *try_to_del_timer_sync* desativa o temporizador caso ele já não esteja executando a sua função, retornando um valor inteiro maior ou igual a zero caso consiga desativar.

Os temporizadores de baixa resolução se baseiam no valor de HZ para determinar com qual frequência eles serão verificados e processados no caso de estarem expirados. No caso do Linux estudado, o valor padrão para HZ é 250, o que faz com que essa verificação seja realizada 250 vezes por segundo, dando uma precisão de 4 milissegundos a estes temporizadores.

A cada 4 milissegundos a fonte de relógio utilizada como relógio global no sistema, causa uma interrupção de *hardware*. Toda interrupção de *hardware* na arquitetura IA-32 é tratada pela função *do_IRQ*, a qual tem seu fluxo de execução demonstrada na figura 3.14. Este fluxo é executado em contexto de interrupção, ou seja, isso tudo ocorre com a maior prioridade de execução no sistema. Desta forma para evitar uma grande interrupção dos processos que estejam executando no sistema, este fluxo executado em contexto de interrupção deve executar o mínimo de tarefas possíveis e o mais rápido possível.

A função *irq_enter* realiza a atualização de algumas estatísticas do sistema. A função *handle_irq* executa a função responsável por tratar a interrupção gerada. No caso as interrupções de temporizadores de baixa resolução são identificadas pela IRQ 0. O tratador desta interrupção para a arquitetura IA-32 é a função *timer_interrupt*, a qual tem seu fluxo

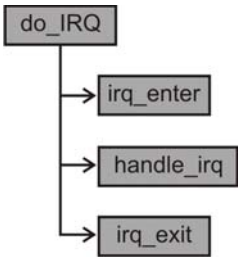

Figura 3.14: Fluxo do tratador de interrupção na arquitetura IA-32 [31]

de execução mostrada na figura 3.15. A função *irq_exit* realiza a saída do contexto de interrupção e executa a função *invoke_softirq* caso necessário. Esta função será discutida em breve.

A função do tratador de eventos do relógio global pode variar, mas sempre executa algo semelhante ao fluxo exibido na figura 3.15, o qual é realizado em máquinas compatíveis com a arquitetura IA-32.

A função *do_timer* é responsável por atualizar a variável *jiffies_64*, o tempo do sistema e calcular as estatísticas de carga do sistema. A função *update_process_times* tem um papel importante na execução dos temporizadores de baixa resolução. A função *account_process_tick* atualiza o tempo de CPU na estrutura do processo. A função *scheduler_tick* atualiza estatísticas específicas do escalonamento do *kernel* e ativa o método de escalonamento periódico da classe de escalonamento responsável pelo processo atual [31].

A função *run_local_timers* é responsável por três tarefas. A primeira é executar as funções dos temporizadores de alta resolução, apenas quando estes estão trabalhando em baixa resolução, como será visto em breve. A segunda é gerar uma interrupção de *software* (*softIRQ*) própria para temporizadores de baixa resolução, definida pela *tag TIMER_SOFTIRQ*. A terceira tarefa é verificar se o *watchdog* está paralisado e resolver o problema.

A *softIRQ* gerada é uma maneira de postergar o trabalho que a interrupção de *hardware* deveria realizar. Desta forma a *softIRQ* realiza as tarefas que demandam um tempo maior de execução em *hard* IRQ, não atrapalhando por um tempo maior que necessário outras interrupções que podem ocorrer e podem estar bloqueadas em *hard* IRQ.

Para inicializar a execução das *softIRQs* pendentes, é chamada a função *invoke_softirq* pela função *irq_exit* fora do contexto de interrupção. Dependendo da configuração do *kernel*, esta função pode executar *do_softirq* com interrupções habilitadas ou não. Uma opção na configuração do *kernel* utilizada neste trabalho é definir a macro *CONFIG_PREEMPT_SOFTIRQS*, a qual é definida por padrão e informa a função *do_softirq* que ela deve executar *trigger_softirqs*. Caso esta macro não tenha sido definida, *do_softirq* executa uma outra função denominada *___do_softirq*, a qual executa todas as *softIRQs* pendentes diretamente.

A macro *CONFIG_PREEMPT_SOFTIRQS* indica que as *softIRQs* devem executar de

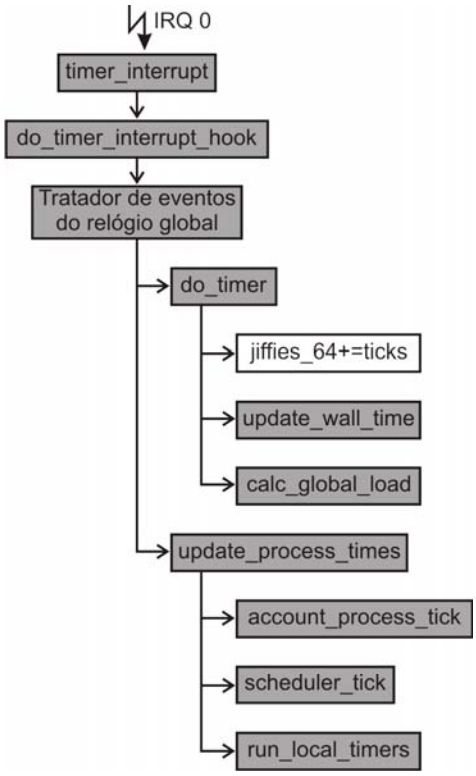

Figura 3.15: Fluxo do tratador de interrupção da IRQ 0 na arquitetura IA-32.

forma que possam ser preemptadas. Para isso existe uma *thread* responsável para executar cada *softIRQ*, as quais executam a função genérica *ksoftirqd* e que recebem como parâmetro a *softirq* que deve executar. Estas *threads* são executadas no sistema com prioridade 49, na classe de escalonamento de tempo real *SCHED_FIFO*. Elas são responsáveis por executar a função atrelada a *softIRQ* a qual corresponde. No caso da *softIRQ* dos temporizadores de baixa resolução, a função é a *run_timer_softirq*, a qual será explicada logo a seguir. Então caso *trigger_softirqs* seja escolhida a executar devido a macro citada anteriormente, esta função acorda todas as *threads* das *softIRQs* pendentes.

Quando *run_timer_softirq* é executada, ela realiza algumas atualizações estatísticas e algumas outras funcionalidades que não é de interesse por enquanto, mas o seu principal objetivo é executar a função *__run_timers*. Esta função executa todos os temporizadores expirados (caso exista algum) e se necessário realiza o cascateamento dos temporizadores, pois estes ficam armazenados na estrutura do já mencionado *Cascading Time Wheel*.

A cada execução de *__run_timers*, é verificado se existe temporizadores no vetor *tv1*, o qual deve conter os que estão mais próximos de expirar. Se for encontrado um ou mais temporizadores na posição relacionada ao valor atual da variável *jiffies*, o sistema retira um a um da lista e executa as funções vinculadas a cada um e também atualiza a variável *timer_jiffies* da estrutura base onde os vetores se localizam.

A cada vez que 256 *ticks* são contabilizados, significa que foram verificadas todas as

posições de *tv1* e tratados todos os temporizadores que existiam nelas. Então o sistema verifica a próxima posição válida para *tv2*, transferindo todos os temporizadores contidos na fila para qual essa posição aponta, para as suas devidas posições no vetor *tv1*. Quando isto ocorrer novamente o sistema verificará a próxima posição do vetor *tv2* e assim por diante. Quando essas transferências ocorrerem 64 vezes, ou seja, todas as posições de *tv2* foram verificadas e todos os seus temporizadores transferidos, então é realizado o mesmo processo com o vetor *tv3*. Assim, é verificada a próxima posição válida no vetor *tv3*, todos os temporizadores contidos na fila correspondente a essa posição são transferidos para preencher tanto o vetor *tv1* quanto o vetor *tv2* e assim sucessivamente, até que todas as posições de *tv3* tenham sido verificadas e os seus temporizadores transferidos.

Ao terminar de esvaziar os vetores *tv1*, *tv2*, *tv3* e quando ocorrer mais um *tick*, ocorrerão então processos semelhantes com o vetor *tv4*, onde cada posição dele será esvaziada para preencher os vetores com faixas de tempo menores. O mesmo ocorre com o vetor *tv5*, sempre que todos os vetores com faixas de tempo menores que a dele estiverem vazios, os temporizadores correspondentes a posição válida em *tv5* são transferidos para os vetores anteriores.

3.4 Temporizadores de Alta Resolução

Os temporizadores clássicos eram responsáveis por prover a contagem do tempo, abrangendo todas as tarefas que necessitavam de tal função e fornecendo a mesma precisão de tempo para todas elas. A abordagem do *kernel* em relação ao gerenciamento destes temporizadores tem um bom desempenho para o caso médio de execução, mas no pior caso seu desempenho não é satisfatório. Devido a esse e outros motivos, muitos estudos foram realizados para desenvolver um temporizador com uma precisão melhor do que o já existente. Estes temporizadores foram denominados de temporizadores de alta resolução (*high resolution timers* - *hrtimers*) e dão precisão temporal na ordem de nanossegundos.

No *kernel* 2.6.16 foi inserida a estrutura básica dos *hrtimers*, a qual proveu a maior parte da sua implementação, menos o suporte a eles. Os temporizadores clássicos passaram a ser implementados no topo dos mecanismos de alta resolução, mas não houve melhorias na sua resolução de tempo [31].

Como existem processos que necessitam de alta precisão temporal, assim como existem as que não precisam de tanta precisão, foi desenvolvido um novo subsistema de tempo para complementar o existente, dividindo os temporizadores em duas categorias. Desta forma os temporizadores clássicos e os temporizadores de alta resolução podem funcionar em paralelo [24]. As categorias são definidas como:

- *Timeouts*: são os temporizadores que requerem baixa precisão e que quase sempre são

excluídos antes mesmo de expirarem [33]. São usados principalmente em tarefas de comunicação de rede e de dispositivos para detectar quando algo não ocorreu como esperado [11] [24].

- *Timers*: são os temporizadores usados para escalonar eventos constantemente, podendo precisar de alta precisão e geralmente expiram. Eles são na maioria das vezes relacionados a aplicações em espaço de usuário ([11] [24].

Desta forma os temporizadores clássicos ou de baixa resolução passaram a ser usados como *timeouts* e os temporizadores de alta resolução (*hrtimers*) classificados como *timers*.

3.4.1 Estruturas de dados

Os temporizadores de alta resolução são organizados por uma estrutura diferente das estruturas dos temporizadores de baixa resolução. Eles são organizados por uma árvore de busca binária balanceada, mais especificamente a árvore vermelha e preta e são ordenados pelo tempo em que devem expirar. Desta forma, evita-se o tempo de resposta de O(n) no pior caso de execução, mantendo um tempo de resposta razoável de O(log(n)) para todas as operações da árvore, onde n é o número de temporizadores a expirarem [17] [24].

Estes temporizadores não se baseiam mais por *ticks* como os clássicos, eles usam o tempo em ordem de nanossegundos. Eles também não são voltados a ter um período de interrupções, mas são programados para gerar interrupções apenas quando necessário (*one-shot timers*). Os *hrtimers* tem como base dois tipos de relógios. O relógio monotônico (*CLOCK_MONOTONIC*) que começa a contar em zero toda vez que o sistema inicia. O outro relógio (*CLOCK_REALTIME*) representa o tempo real do sistema, podendo apresentar saltos de tempo, caso o tempo seja alterado por ajustes [13] [31].

```
<linux/hrtimer.h>

struct hrtimer_cpu_base {
  atomic_spinlock_t          lock;
  struct hrtimer_clock_base  clock_base[HRTIMER_MAX_CLOCK_BASES];
#ifdef CONFIG_HIGH_RES_TIMERS
  ktime_t                    expires_next;
  int                        hres_active;
  unsigned long              nr_events;
#endif
#ifdef CONFIG_PREEMPT_SOFTIRQS
  wait_queue_head_t          wait;
#endif
};
```

Figura 3.16: Estrutura de registro de CPU para *hrtimer*.

O *kernel* possui uma estrutura de dados para registro de uma base para os *hrtimers* por CPU, podendo desta forma, organizar melhor os temporizadores de alta resolução por CPU. A estrutura é apresentada na figura 3.16 e seus campos são definidos a seguir:

- *lock*: É uma variável de bloqueio que garante acesso único a base, protegendo assim a base da CPU, as bases de relógio associadas a CPU e os temporizadores.
- *clock_base*: Vetor com as bases de relógio para esta CPU. *HRTIMER_MAX_CLOCK_BASES* é definido com valor igual a 2, indicando a base de relógio monotônica e a base de tempo real.
- *expires_next*: Informa o tempo absoluto do próximo evento que deve ocorrer.
- *hres_active*: Informa o estado do modo de alta resolução, se está ativo ou se apenas o modo de baixa resolução está disponível. Usada como uma variável *boolean*.
- *nr_events*: Informa o número total de eventos de interrupção de tempo.
- *wait*: É uma fila de espera (*wait queue*), usada quando necessário excluir um temporizador e ele estiver no estado de execução de sua função. Então o temporizador é inserido na fila de espera (a não ser em alguns casos especiais) e a função chamada para excluir o temporizador dorme enquanto a função executa.

Todas as bases de relógio para temporizadores de tempo real são também registrados por CPU. Como existem as bases de relógio monotônico e a de tempo real, estas são registradas uma por CPU, pois cada uma possui dados que diferenciam entre elas. A estrutura responsável por armazenar estes dados podem ser verificadas na figura 3.17.

```
<linux/hrtimer.h>

struct hrtimer_clock_base {
  struct hrtimer_cpu_base   *cpu_base;
  clockid_t                 index;
  struct rb_root            active;
  struct list_head          expired;
  struct rb_node            *first;
  ktime_t                   resolution;
  ktime_t                   (*get_time)(void);
  ktime_t                   softirq_time;
#ifdef CONFIG_HIGH_RES_TIMERS
  ktime_t                   offset;
#endif
};
```

Figura 3.17: Estrutura da base de relógio para *hrtimer*.

Os campos da estrutura *hrtimer_clock_base* são definidos a seguir:

- *cpu_base*: É um ponteiro para a base de CPU onde esta base de relógio está registrada, já que cada uma delas é registrada uma vez em cada CPU.
- *index*: Identifica a qual base de relógio se refere (*CLOCK_REALTIME* ou *CLOCK_MONOTONIC*). Utilizada para identificar a qual base de relógio o temporizador pertence quando é necessário transferi-lo para outra CPU.
- *active*: É o nodo principal da árvore vermelha e preta, a qual ordena os temporizadores ativos.
- *expired*: É uma lista para armazenar os temporizadores expirados se necessário.
- *first*: É um ponteiro que indica o próximo temporizador que irá expirar.
- *resolution*: Informa a resolução do relógio em nanossegundos.
- *get_time*: Ponteiro para uma função que retorna o tempo atual do relógio.
- *softirq_time*: Informa o tempo de execução da fila de *hrtimer* em *softIRQ*.
- *offset*: É o tempo de compensação para o relógio na base monotônica.

O *kernel* define uma variável global por CPU do tipo *hrtimer_cpu_base* chamada *hrtimer_bases*, a qual tem o seu vetor *clock_base* inicializada pelo sistema como mostra a figura 3.18. As duas bases do relógio são inicializadas no modo de baixa resolução (*KTIME_LOW_RES*), pois de início o sistema apenas suporta este modo de resolução.

```
<kernel/hrtimer.c>

DEFINE_PER_CPU(struct hrtimer_cpu_base, hrtimer_bases) =
{
  .clock_base =
  {
    {
      .index = CLOCK_REALTIME,
      .get_time = &ktime_get_real,
      .resolution = KTIME_LOW_RES,
    },
    {
      .index = CLOCK_MONOTONIC,
      .get_time = &ktime_get,
      .resolution = KTIME_LOW_RES,
    },
  }
};
```

Figura 3.18: Inicialização das bases de relógio por CPU da variável *hrtimer_bases*.

A estrutura responsável por definir um temporizador de alta resolução no sistema é demonstrada na figura 3.19. E seus campos podem ser definidor como:

```
<linux/hrtimer.h>

struct hrtimer {
  struct rb_node              node;
  ktime_t                     _expires;
  ktime_t                     _softexpires;
  enum hrtimer_restart        (*function)(struct hrtimer *);
  struct hrtimer_clock_base   *base;
  unsigned long               state;
  struct list_head            cb_entry;
  int                         irqsafe;
#ifdef CONFIG_TIMER_STATS
  int                         start_pid;
  void                        *start_site;
  char                        start_comm[16];
#endif
};
```

Figura 3.19: Estrutura do hrtimer.

- *node*: Nodo da árvore vermelho e preta, utilizado para inserir o temporizador na árvore ordenada.

- *_expires*: Informa o tempo absoluto em que o temporizador deve expirar na representação interna de *hrtimers*. O tempo é relacionado ao relógio em que o temporizador é baseado. Ele é configurado adicionando folga de tempo em relação ao valor de *_softexpires*. Desta forma o temporizador pode ser executado sem problemas até este valor de tempo.

- *_softexpires*: Informa o tempo absoluto a partir de quando o temporizador já pode expirar. Desta forma se o sistema estiver executando um temporizador que expirou, ele realiza uma verificação procurando temporizadores que já estiverem expirados em relação ao seu valor de *_softexpires* e os executam, mesmo que eles ainda não tenham expirado em relação ao seu valor de *_expires*. Isso evita interrupções desnecessárias do processador.

- *function*: É um ponteiro para a função que deve ser executada quando o temporizador expira. Esta função pode retornar dois valores (*HRTIMER_RESTART* ou *HRTIMER_NORESTART*) quando é executada, informando se o temporizador deve ser reiniciado ou não.

- *base*: É um ponteiro para a base do relógio. Lembrando que por este ponteiro pode-se diferenciar em qual CPU e qual base de relógio (monotônico ou tempo real) o temporizador está.

- *state*: Informa o estado do temporizador. Ele pode ser definido como inativo (*HRTIMER_STATE_INACTIVE*), enfileirado na árvore vermelha e preta (*HRTIMER_STA-*

TE_ENQUEUED), executando sua chamada de função (*HRTIMER_STATE_CALLBACK*) ou que ele está migrando para outra CPU (*HRTIMER_STATE_MIGRATE*).

- *cb_entry*: É a cabeça de uma lista, usada anteriormente para enfileirar os temporizadores que expiravam. Essa variável não é mais utilizada na versão do *kernel* estudada, mas a partir da versão 2.6.32-rc ela já foi retirada da estrutura.
- *irqsafe*: Indica que o temporizador não vai ser executado através de *softIRQ*, sendo executada assim diretamente em contexto de interrupção (*hard* IRQ). Seus valores são 1 ou 0.
- *start_pid*, *start_site*, *start_comm*: São campos de estatística do temporizador. O primeiro informa o identificador *pid* do processo que inicializou o temporizador. O segundo guarda o lugar onde o temporizador foi iniciado. O último informa o nome do processo que iniciou o temporizador.

Uma das utilidades para temporizadores de alta resolução é a de contar com precisão o tempo que um processo deve dormir. Para esta aplicação o *kernel* disponibiliza uma estrutura de dados específica, demonstrada na figura 3.20. Ela possui como campos, um temporizador de alta resolução e um ponteiro para o processo que deve ser acordado no determinado tempo.

```
<linux/hrtimer.h>

struct hrtimer_sleeper {
  struct hrtimer        timer;
  struct task_struct    *task;
};
```

Figura 3.20: Estrutura do *hrtimer_sleeper*.

3.4.2 Utilização dos Temporizadores de Alta Resolução

Como nos temporizadores de baixa resolução, os de alta resolução também possuem funções para sua utilização. Começando pelas funções responsáveis por iniciar um temporizador de alta resolução, as quais podem ser verificadas na figura 3.21.

A função *hrtimer_init* configura alguns dados de um temporizador de alta resolução, o qual é informado através do ponteiro *timer*. A variável *clock_id* representa sob qual base de relógio o temporizador deve ser registrado, o *CLOCK_REALTIME* ou *CLOCK_MONOTONIC*. Enquanto que a variável *mode* representa em qual modo o valor de tempo deve ser representado, que pode ser no modo de tempo relativo (*HRTIMER_MODE_REL*) ou absoluto (*HRTIMER_MODE_ABS*). A função que o temporizador deve executar quando expirar deve ser informada diretamente antes de chamar uma função para ativar o temporizador na CPU [13].

```
<linux/hrtimer.h>

void hrtimer_init(struct hrtimer *timer, clockid_t which_clock,
                  enum hrtimer_mode mode);

int hrtimer_start(struct hrtimer *timer, ktime_t tim,
                  const enum hrtimer_mode mode);

int hrtimer_start_range_ns(struct hrtimer *timer, ktime_t tim,
             unsigned long range_ns, const enum hrtimer_mode mode);

int hrtimer_restart(struct hrtimer *timer);
```

Figura 3.21: Funções utilizadas para configurar e ativar um *hrtimer*.

A função *hrtimer_start_range_ns* é utilizada para configurar o tempo do temporizador e para ativá-lo no sistema. Importante ressaltar que esta função é utilizada tanto para ativar um temporizador pela primeira vez como para reativar um já existente. A variável *tim* informa o tempo em que o temporizador deve expirar, *mode* é equivalente a variável *mode* da função *hrtimer_init* e *range_ns* informa uma variação de tempo para ser acrescido ao tempo da variável *tim*, criando a folga de tempo possível entre *_expires* e *_softexpires* da estrutura *hrtimer*.

A função *hrtimer_start* tem a mesma função de *hrtimer_start_range_ns* com a única diferença que ela não recebe valor de folga *range_ns*. Desta forma o temporizador deve expirar no tempo informado pela variável *tim* [13]. A função *hrtimer_restart* é utilizada para reativar um temporizador de alta resolução, o qual é informado através do ponteiro *timer*. Ela supõe que o temporizador já está com os valores que devem expirar corretos e o reativa utilizando tais valores.

```
<linux/hrtimer.h>

int hrtimer_cancel(struct hrtimer *timer);

int hrtimer_try_to_cancel(struct hrtimer *timer);
```

Figura 3.22: Funções para cancelar um *hrtimer*.

As funções utilizadas para cancelar um temporizador de alta resolução podem ser verificadas na figura 3.22. Para cancelar um temporizador já iniciado, são fornecidas duas funções que diferem um pouco uma da outra. A função *hrtimer_try_to_cancel* tenta cancelar o temporizador informado pelo ponteiro *timer*. Ela verifica se o temporizador já está executando sua função, neste caso não cancela o temporizador, retornando o valor -1 como resultado. Caso o temporizador estiver ativo e não executando sua função, ele é cancelado e a função retorna o valor 1, caso ele já esteja inativo, nada é feito com ele, a função apenas retorna o valor 0 [13].

A função *hrtimer_cancel* retorna o valor 0 caso o temporizador já esteja inativo. Caso o temporizador esteja ativo e executando sua função, *hrtimer_cancel* espera o término dela através da função *hrtimer_wait_for_timer*, que utiliza-se da variável *wait* da base de CPU para esperar o processador sem grande *overhead*. E quando é cancelado o temporizador que estava ativo, é retornado o valor 1.

```
<linux/hrtimer.h>

int schedule_hrtimeout_range(ktime_t *expires, unsigned long delta,
                             const enum hrtimer_mode mode);

int schedule_hrtimeout(ktime_t *expires,
                       const enum hrtimer_mode mode);
```

Figura 3.23: Funções de *sleep* que utilizam *hrtimers*.

Uma das utilizações dos temporizadores de alta resolução é fazer um processo dormir por um período de tempo. Existem duas funções para isto que se utilizam da estrutura *hrtimer_sleeper* apresentada na figura 3.20. Estas funções podem ser verificadas na figura 3.23.

A função *schedule_hrtimerout_range* recebe como parâmetros o tempo que o processo deve dormir, representado pelo ponteiro *expires*, uma variável *delta* que define um tempo de folga para o temporizador, como explicado anteriormente, e uma variável *mode* que informa se o tempo de *expires* está no modo absoluto ou relativo [16]. Esta função configura um *hrtimer_sleeper* para acordar o processo que o chamou no período de tempo determinado pelos parâmetros informados. Caso o tempo informado em que o temporizador deve expirar seja menor que o tempo atual, o processo é acordado imediatamente.

A função *schedule_hrtimerout* é semelhante a função *schedule_hrtimerout_range*, com a única diferença que ela não disponibiliza uma faixa de tempo onde o processo pode ser acordado, mas apenas o tempo certo no qual ele deve ser acordado.

3.4.3 Funcionamento dos *hrtimers* em Baixa Resolução

Os temporizadores de alta resolução nem sempre trabalham em alta resolução temporal, ou seja, nem sempre eles possuem uma resolução de nanossegundos. Eles podem trabalhar no modo de baixa resolução por alguns motivos, como o sistema não possuir nenhuma fonte de relógio que possa prover a resolução necessária ou quando o sistema operacional está iniciando e ainda não inicializou a fonte de relógio necessária. Então quando estes temporizadores trabalham em modo de baixa resolução, eles trabalham na mesma frequência de HZ, tendo esta variável como padrão igual a 250, a resolução é de 4 milissegundos.

Como explicado anteriormente, foi mostrado como funcionam os temporizadores de

baixa resolução e o que ocorre a cada interrupção dos temporizadores. A cada interrupção destes é executada a função *run_local_timers*, a qual gera interrupções de *software* para executar os temporizadores clássicos, mas também chama a função *hrtimer_run_queues*, sendo esta responsável por executar os temporizadores *hrtimers* no modo de baixa resolução.

A função *hrtimer_run_queues* executa em contexto de interrupção (*hard* IRQ). Ela verifica todos os temporizadores expirados de todas as bases de relógio da CPU que gerou a interrupção e trata eles de duas formas distintas:

- Os temporizadores que tiverem sua variável *irqsafe* configurada com o valor 1, terão suas funções executadas imediatamente através da função *__run_hrtimer* ainda em *hard* IRQ;

- Os demais temporizadores expirados serão adicionados na lista *expired* da base de relógio, tendo suas funções executadas em outro momento através da *softIRQ* responsável pelos temporizadores de alta resolução (*HRTIMER_SOFTIRQ*).

A função termina sua execução acordando a *thread* de *softIRQ* (*ksoftirqd*) para postergar a execução dos temporizadores, caso tenha algum temporizador expirado que tenha sido incluído na lista *expired*.

Então, como os *hrtimers* podem ser executados de duas formas diferentes, eles precisam de duas funções distintas. No caso da função *__run_hrtimer*, apenas um *hrtimer* é executado a cada chamada dela. Onde o temporizador tem sua função executada ela pode retornar um valor *HRTIMER_RESTART*, indicando que este mesmo temporizador deve ser reiniciado, voltando a ser inserido na base a qual ele pertencia.

No caso dos temporizadores executados pela *thread* de *softIRQ*, eles devem esperar pela execução da *thread ksoftirqd*, a qual possui prioridade 49 e pertence a classe de escalonamento *SCHED_FIFO* que segue o mesmo processo já explicado na seção de funcionamento dos temporizadores de baixa resolução. Ela chamará a função *run_hrtimer_softirq* e esta função (no *kernel* estudado) chama outra (*hrtimer_rt _run_pending*), a qual é a verdadeira responsável por executar todos os temporizadores de alta resolução que estão na lista *expired*. Da mesma forma que em *__run_hrtimer*, os temporizadores que necessitam ser reinicializados, também serão por esta função. Por fim ele acorda os processos que estão registrados na fila de espera *wait* da base de CPU através da função *wake_up_timer_waiters*. Estes processos devem estar esperando para terminar a exclusão de temporizadores, os quais quando estavam executando suas funções, tiveram que ser excluídos durante este processo, fazendo com que a tarefa de exclusão espere pelo fim da execução de sua tarefa.

3.4.4 Funcionamento dos *hrtimers* em Alta Resolução

Enquanto os temporizadores de alta resolução estiverem executando em modo de baixa resolução, a cada *tick* é verificado se eles já podem trabalhar em alta resolução através da função *hrtimer_run_pending* que é chamada a cada execução de *run_timer_softirq*. Se verificado que pode ocorrer a troca para alta resolução, a função *hrtimer_switch_to_hres* realiza esta troca. Assim, os temporizadores de alta resolução funcionarão realmente em alta resolução.

O dispositivo de eventos utilizado para obter a alta resolução, trabalha em modo *one-shot* e não periódico. A cada interrupção gerada, esse dispositivo chama a função *hrtimer_interrupt* em contexto de interrupção. Essa função é semelhante a *hrtimer_run_queues*, a qual foi explicada anteriormente, mas com algumas diferenças necessárias.

A função *hrtimer_interrupt* verifica quais os temporizadores que estão expirando no momento, em cada base de relógio (monotônica e tempo real) e através de sua variável *irqsafe* verifica se ele deve ter sua função executada imediatamente ainda em *hard* IRQ, ou se ela deve ser postergada através de *softIRQ*, da mesma forma como explicado anteriormente na execução da função *hrtimer_run_queues*. Quando isso tiver sido realizado para todos os temporizadores expirados, a função calcula quando deve ocorrer a próxima interrupção do dispositivo de eventos e tenta por cinco vezes o reprogramar. Caso não consiga reprogramar, supõe-se que ele esteja se suspendendo e altera-se o valor de *min_delta_ns* do dispositivo de eventos para três vezes o valor gasto executando a função *hrtimer_interrupt*, forçando os temporizadores programarem suas interrupções baseados nesse valor mínimo. Assim o dispositivo de eventos tem um tempo maior entre interrupções para evitar essas suspensões.

Por fim caso algum temporizador tenha sido incluído na lista *expired* da base de relógio, a função acorda a *thread* de *softIRQ* (*ksoftirqd*) responsável por executar os temporizadores de alta resolução. Assim como explicado anteriormente sobre essa *thread*, ela é escalonada pelo processador com prioridade 49, pela classe de escalonamento de tempo real *SCHED_FIFO*.

3.5 *Ticks* Dinâmicos

Como visto anteriormente, os temporizadores de baixa resolução trabalham baseados em *ticks* periódicos. Assim o dispositivo de evento de relógio é programado para gerar uma interrupção a cada período determinado de tempo. Nem sempre existem temporizadores a serem processados a cada *tick*, mas mesmo assim são geradas interrupções. Isso causa ineficiência, pois são executados trabalhos desnecessários e evita economia de energia, a qual é necessária para alguns sistemas.

Para resolver este problema foi desenvolvido um método de *ticks* dinâmicos ou também conhecido como *tickless*, os quais só geram *ticks* quando a CPU precisar contar a passagem

deles. Desta forma o sistema pode entrar por períodos de inatividade maiores, economizando mais energia. Mas estes *ticks* dinâmicos são ativados apenas se a fonte de relógio do sistema puder gerar interrupções do tipo *one-shot* e se eles forem definidos em tempo de compilação do *kernel*.

Assim os *ticks* são gerados dinamicamente de forma periódica quando necessário, ou seja, eles são ativados apenas quando algum processo está em execução, de outra forma eles são desativados, sendo reativados quando ocorre uma interrupção externa ou no momento em que o próximo *tick* relevante deva expirar. Para identificar quando nenhum processo está executando e desativar os *ticks* periódicos, utiliza-se de um processo especial chamada "*idle task*", a qual é escalonada para executar quando não existe nenhuma outra a ser executada. Ela não realiza tarefa alguma, é utilizada apenas para informar que o sistema está ocioso [31].

Como o sistema inicia com os temporizadores em modo de baixa resolução, a cada *tick* do sistema é verificado se os temporizadores de alta resolução ou os *ticks* dinâmicos podem ser habilitados e caso possam, eles são habilitados. Os *ticks* dinâmicos só podem ser habilitados com temporizadores de baixa resolução sob duas condições: um dispositivo de evento de relógio que suporte o modo *one-shot* deve estar disponível e os temporizadores de alta resolução não podem estar habilitados. No caso dos temporizadores usados sejam os de alta resolução, os *ticks* dinâmicos são habilitados de forma mais fácil [31].

Para habilitar os *ticks* dinâmicos em modo de baixa resolução é necessário configurar o dispositivo de eventos de relógio para o modo *one-shot* e instalar um tratador de *tick* apropriado. Caso o sistema esteja em modo de alta resolução, o dispositivo de evento de relógio já estará em modo *one-shot* e facilita esta habilitação.

O tratador instalado deve realizar todas as ações necessárias para o mecanismo de *tick* e reprogramar o dispositivo de *tick* de forma que o próximo *tick* expire no tempo correto. Ele é responsável por verificar qual CPU no sistema será responsável por gerar o *tick* global. Diferente de como acontecia anteriormente com os *ticks* periódicos, as CPUs podem entrar em grandes períodos de inatividade. Sendo assim, o *tick* global não é responsabilidade de apenas uma CPU o tempo todo. Caso a CPU responsável pelo *tick* global vá dormir por um período longo de tempo, ela abre mão desta tarefa e a próxima CPU que tiver seu tratador de *tick* chamado fica como responsável, assim esta tarefa é revesada entre as CPUs.

Existe também a possibilidade de todas as CPUs dormirem ao mesmo tempo por um longo período. Desta forma, a primeira CPU que acordar atualiza o *tick* global na quantidade exata de *jiffies* que ela dormiu. Esse número de *jiffies* é calculado através do tempo atual do relógio e o tempo que a CPU registrou quando começou a dormir.

Capítulo 4

Inversão de Prioridade Causada por Temporizadores de Alta Resolução

Nos capítulos anteriores foram apresentados a teoria básica sobre processos de um sistema operacional, interrupções no Linux e como o *kernel* do Linux funciona em relação ao escalonamento de processos de tempo real e seus temporizadores. Neste capítulo será realizada uma análise detalhada da inversão de prioridades que pode acontecer em determinadas ocasiões.

4.1 Caracterização do Problema

Os temporizadores de alta resolução foram um grande passo para o aprimoramento do *kernel* em relação ao seu escalonamento de processos. Devido a eles, os processos podem ser escalonados com precisão maior de tempo. Como visto, a execução destes temporizadores pode acontecer com maior ou menor precisão devido ao momento em que ele é processado, podendo ser executado diretamente em *hard* IRQ ou pode ter seu trabalho postergado através de *softIRQ*. Esta técnica de postergação utilizada no Linux e denominada *softIRQ* tem o objetivo de fazer com que a execução destes processos atrapalhe o mínimo possível os que estejam executando no sistema [38].

Os temporizadores em *software* são criados indiretamente por qualquer processo, utilizando funções e aplicações através da *interface* de programação de aplicações (API - *Application Programming Interface*). Esta API dos temporizadores possibilita a criação e manipulação deles de uma forma segura ao sistema operacional, não permitindo que os temporizadores sejam criados de forma que obviamente irá diminuir muito o desempenho do sistema ou até travá-lo. Assim, qualquer processo executando no sistema pode chamar uma função do sistema que crie um ou mais de um temporizador para contar o tempo com alguma finalidade.

No Linux, a maioria dos temporizadores de alta resolução são executados em *hard* IRQ, como pode ser verificado na figura 4.1, a qual foi gerada através de dados obtidos do *kernel* do Linux estudado. Para obter todos os dados utilizados neste trabalho, foram criados vários processos através de alguns programas de uso normal no Linux, principalmente os que se utilizam bastante de acesso a disco e rede, para assim tentar simular um ambiente onde o sistema tenha alta carga de processos e temporizadores de alta resolução. Os tempos obtidos neste trabalho são medidos utilizando a estrutura *ktimer* e as medições possuem resolução de nanossegundos.

Como a maioria dos temporizadores de alta resolução criados são executados em *hard* IRQ, ou seja, com a maior precisão possível, isto faz com que os processos do sistema sofram bastante com interferências causadas por eles. Estes temporizadores são utilizados por vários processos que necessitam de precisão temporal. Uma utilização muito frequente deles é de servir como *sleep*, ou seja, fazer com que um processo durma por um determinado tempo, criando um temporizador para tirar este processo da fila de espera para a fila de prontos (conhecido também como ato de acordar um processo) assim que seu tempo expirar.

Para a utilização de *sleeps*, o *kernel* disponibiliza a estrutura específica *hrtimer_sleeper*, juntamente com duas funções para a utilização desta estrutura como explicado anteriormente. Depois de pesquisas realizadas no kernel estudado, constatou-se que todos os temporizadores utilizados como *sleeps* através desta estrutura são executados em *hard* IRQ. Através da figura 4.1 pode-se verificar a diferença entre as quantidades de temporizadores de alta resolução executados no sistema. Nesta figura realiza-se a contagem de *hrtimers* executados por segundo, dividindo-os em três formas de execução, sendo os executados em *SoftIRQ* (que pode-se ver no gráfico que a maioria das vezes não existe nenhum temporizador executado por segundo), os temporizadores executados em *hard IRQ* e que são utilizados como *sleeps* (que pode-se verificar que uma boa quantidade de vezes chega a executar cerca de 310 *hrtimers* por segundo) e os demais executados em *hard IRQ* (que pode-se verificar que mesmo executando bastante temporizadores por segundo, ainda executam menos que os utilizados como *sleeps*). O gráfico da figura 4.1 foi obtido através de uma medição de 10 minutos, o que levou a 600 repetições de medições da quantidade de hrtimers executados no sistema, uma por segundo, quando o sistema executava cerca de 40 processoss (sem considerar seus processos filhos).

No *kernel* do Linux estudado neste trabalho, quando um processo normal está na fila de espera e volta para a fila de prontos, ele volta a ter uma fatia de tempo da CPU de acordo com sua prioridade. Mas quando um processo de tempo real está na fila de espera e volta para a fila de prontos, este só será executado caso sua prioridade seja maior que os outros processos de tempo real executando na CPU. Como os processos de tempo real no Linux têm prioridade maior do que os processos normais do sistema, mesmo que existam vários processos normais na fila de prontos, se existir pelo menos um processo de tempo real nesta fila, seja qual for sua prioridade, ele será executado antes de todos os outros processos normais.

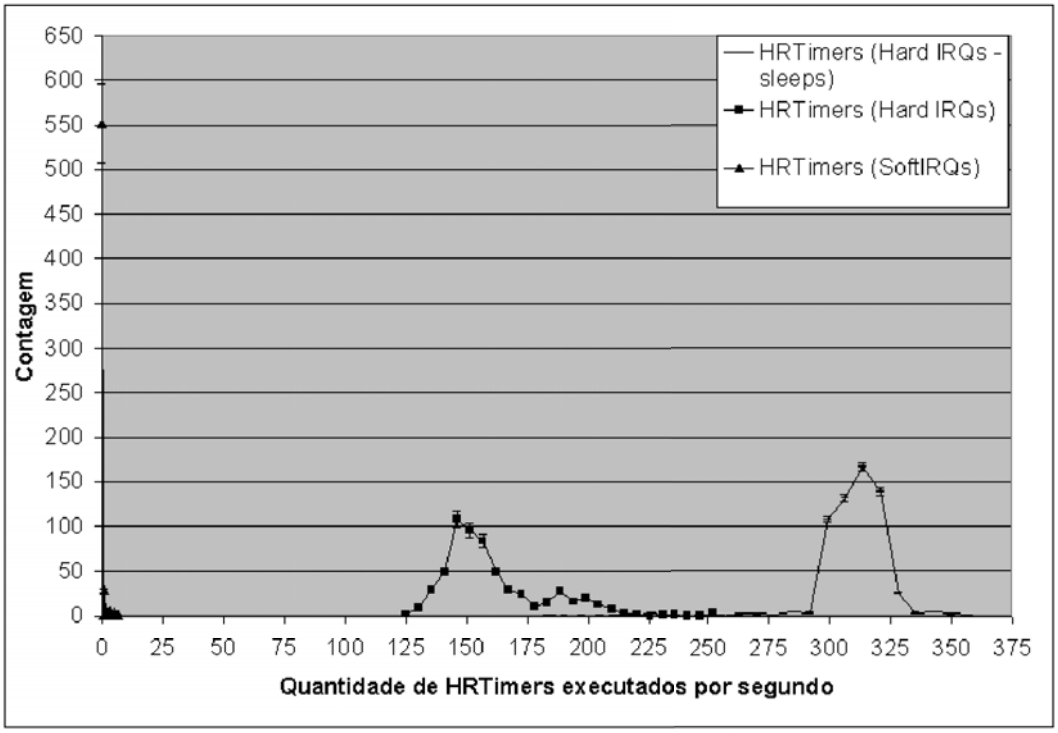

Figura 4.1: Diferença entre as quantidades de *HRTimers* executados no sistema.

Processos de tempo real devem ter suas prioridades respeitadas, pois caso contrário, seus *deadlines* correm um risco maior de não poderem ser cumpridos. Estudos relacionados a processos de tempo real e suas prioridades são muito comuns, como por exemplo, estudos que propõem métodos para resolver inversões de prioridades em processos de tempo real [29].

No *kernel* estudado, muitos processos utilizam-se dos temporizadores de tempo real para dormirem por um breve momento, cumprindo assim seus períodos de execução, voltando para a fila de prontos da CPU nos momentos corretos, onde podem ser executados novamente de acordo com as decisões do escalonador. Tais processos, quando devem voltar à fila de prontos, são processados pelos temporizadores diretamente em *hard* IRQ, interrompendo assim qualquer processo que esteja sendo executado pelo processador. Então, caso existam processos de tempo real sendo executados, eles serão interrompidos por qualquer processo que deva voltar para a fila de prontos, podendo ser ele até o processo menos relevante do sistema (excluindo destes o processo especial *Idle*).

Os processos que são acordados pelos temporizadores entram na fila de prontos, mas caso eles não possuam prioridade maior a todos os processos de tempo real que já estejam na fila de prontos ou executando, eles têm que esperar até que todos os processos de maior prioridade executem, para assim o escalonador poder processá-los. Então, a interferência causada por estes processos devido a execução do seu temporizador em relação aos processos de tempo real é desnecessária.

Para entender melhor o que ocorre no Linux estudado, supõe-se três processos de tempo real (T1, T2 e T3) que devem ser escalonados por um processador através da classe de escalonamento *SCHED_FIFO*. O processo T1 de prioridade 99 possui um período de execução e *deadline* igual a 50 e tempo de execução igual a 15. Os processos T2 e T3 têm prioridades 2 e 1 respectivamente, tendo os seus períodos e *deadlines* iguais a 46. Na representação do escalonamento destes processos o tratador de interrupção é representado por um processo de mais alta prioridade, pois ele é tratado em *hard* IRQ e interrompe qualquer processo executando no sistema. A figura 4.2 mostra a situação de forma simplificada.

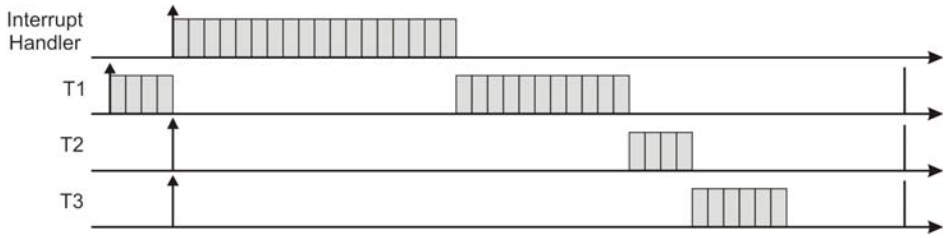

Figura 4.2: Exemplo de escalonamento de processos no Linux estudado.

Como pode-se ver na figura 4.2, o processo T1 é o primeiro a chegar à fila de prontos e começa a executar. Logo após, o temporizador de alta resolução responsável por acordar os processos T2 e T3 expira e o tratador de interrupção entra em execução, interrompendo o processo T1 por um tempo considerável. Quando o tratador termina o seu trabalho, o processo T1 volta a executar, pois ele ainda é o processo de maior prioridade na fila de prontos. Assim que T1 termina sua execução, T2 e T3 podem executar de acordo com suas prioridades. Desta forma verifica-se como os temporizadores de alta resolução interferem na execução de alguns processos, realizando um trabalho desnecessário para o momento, pois acorda os processos de menor prioridade que não executarão ainda, causando uma inversão de prioridades.

4.2 Medições de Interferência

A interferência que o tratador de interrupções pode causar sobre os processos de mais alta prioridade pode ser calculada, podendo-se assim ter uma estimativa de quanto estas interrupções podem prejudicar os processos de tempo real. Através da equação (4.1) pode-se calcular o tempo que o tratador de interrupções gasta para processar os temporizadores expirados.

$$C_T = C_{TC} + C_{FG} + C_{TAR} \tag{4.1}$$

As variáveis da equação (4.1) apresentada podem ser definidas como:

- C_T: Tempo de execução gasto pelo tratador de interrupções para processar todos os temporizadores de alta resolução expirados;
- C_{TC}: Tempo gasto nas trocas de contexto entre o processo que estava executando e o código do tratador de interrupções;
- C_{FG}: Tempo gasto para o tratador de interrupções processar todas as funções gerais da sua rotina, como atualizar estatísticas, variáveis entre outras tarefas;
- C_{TAR}: Tempo gasto para processar todos os temporizadores de alta resolução expirados.

A variável C_{TAR} é um pouco mais complexa que as outras, sendo calculada através da somatória mostrada na equação (4.2):

$$C_{TAR} = \sum_{i=1}^{n} \Big(C_{FT}(i) \Big) \tag{4.2}$$

A equação (4.2) é o cálculo de C_{TAR}, o qual é o somatório das variáveis $C_{FT}(i)$, onde i varia de 1 até n, sendo n o número de temporizadores de alta resolução expirados e C_{FT} representa o tempo de execução da função ligada ao temporizador, o qual pode variar devido a alguns fatores. Por exemplo, quando a função tenta acordar um processo que já está na fila de prontos, não é necessário muito tempo de processamento. Quando deve-se acordar um processo que está realmente dormindo, ainda pode existir variações no tempo de execução devido a situações diferentes em que o processo pode se encontrar, como também problemas de cache e outros fatores comuns de um sistema operacional. Estas variações de tempo podem ser verificadas na figura 4.3.

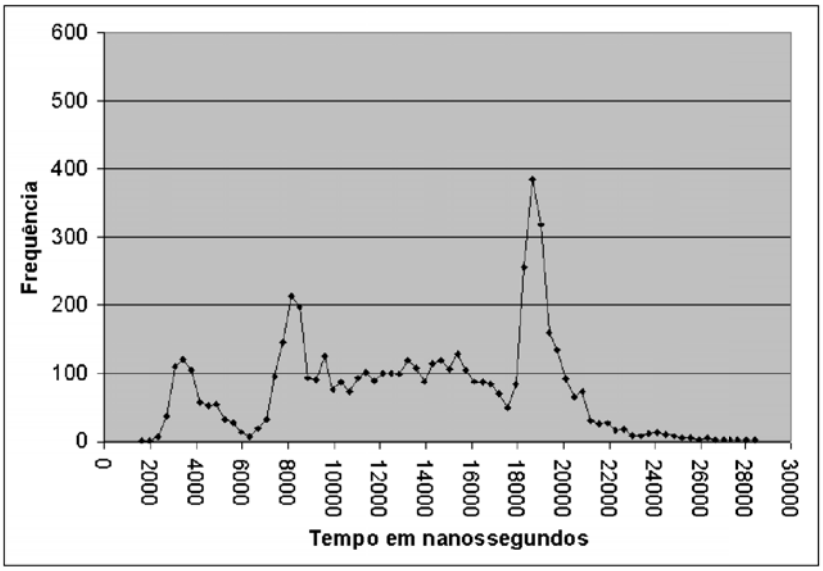

Figura 4.3: Variação de tempo para acordar um processo através do tratador de interrupção.

Para validar o cálculo da equação (4.1) foram criados dois conjuntos de processos, para assim medir o tempo de execução de cada variável necessária na equação e depois comparar com o resultado final do processamento dos processos. Para facilitar e melhorar a precisão das medições, todos os processos utilizados são escalonados pela política de escalonamento de tempo real *SCHED_FIFO*. Os dados do primeiro conjunto de processos foram definidos como apresentado na tabela 4.1.

Processo	**Prioridade**	**Período**	***Deadline***	**Tempo de execução**
TA	99	170 us	170 us	20 us
TB	5	170 us	170 us	20 us

Tabela 4.1: Dados dos processos TA e TB.

O conjunto de processos apresentado tem seus tempos de execução bem pequenos, executando em questão de microssegundos, assim como seus períodos e *deadlines*. Suas prioridades são diferentes, lembrando que a prioridade 99 é a mais alta e 1 é a mais baixa. Na execução medida no *kernel* foi induzido que o processo TB dormisse e só acordasse durante a execução do processo TA. Desta forma, o processo TA foi iniciado e 5104 nanossegundos depois ocorreu uma interrupção, a qual foi responsável por tratar o temporizador de alta resolução que acordou o processo de tempo real TB. Através das medições realizadas no *kernel* foi criada a tabela 4.2 que possui os valores necessários para se aplicar às equações (4.1) e (4.2). Todas as medições dos tempos de execução dos processos no *kernel* foram realizadas repetidamente cerca de vinte vezes, então como os valores não variaram tanto entre as medições, para cada caso foi utilizada uma das medições que tenha se aproximado mais da média entre as medições realizadas.

Variável	**Valor**
C_{TC}	1487 ns
C_{FG}	1692 ns
$C_{FT}(1)$"TB"	12197 ns

Tabela 4.2: Tempos do primeiro conjunto de processos (TA e TB).

Aplicando os valores da tabela 4.2 às equações (4.2) e (4.1) respectivamente, obtem-se o tempo de interferência que o processo TA sofre por causa do processo TB. Este cálculo pode ser verificado na equação (4.3), já a figura 4.4 mostra como ocorreu a primeira execução dos processos TA e TB.

$$\begin{aligned} C_{TAR} &= 12197ns \\ C_T &= 1487 + 1692 + 12197 \\ C_T &= 15376ns \end{aligned} \tag{4.3}$$

Na execução deste conjunto de processos representado pela figura 4.4, são apresentados três processos (TA, TB e TI), onde TI é apenas uma forma de representar a execução do tratador de interrupção, o qual é utilizado para executar os temporizadores de alta resolução expirados. No caso, o processo TA é interrompido por TI para que o processo TB seja acordado. Como TB tem prioridade menor que TA, assim que TI termina de executar, o escalonador de processos do *kernel* não permite que TB seja executado até que TA tenha terminado sua execução. Desta forma, TA demorou mais para concluir sua execução, para que o tratador de interrupção acorde o processo TB com a precisão que os temporizadores de alta resolução oferecem. Vale ressaltar que, devido as medições serem na ordem de nanossegundos, a execução dos processos varia um pouco, como pode ser notado no exemplo da figura 4.4, onde os processos que deveriam executar em 20000 nanossegundos, acabam por executar em um tempo um pouco maior.

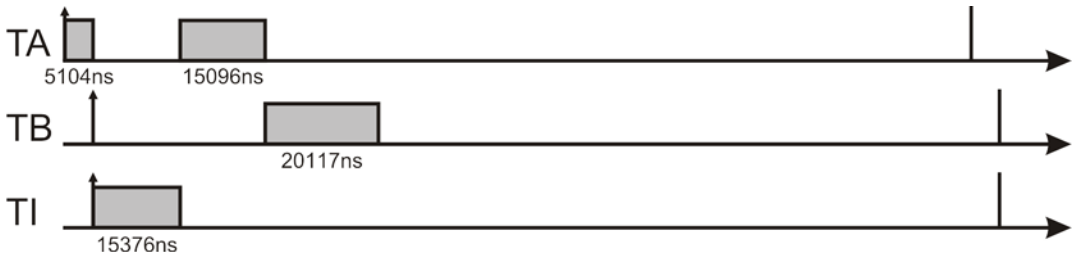

Figura 4.4: Representação da execução dos processos TA e TB.

Para mostrar como o tempo de interferência que o processo TA sofre pode ser mais significativo, é mostrado outro conjunto de processos semelhante ao primeiro conjunto, mas com mais processos (TA, TB, TC, TD e TE). Os dados destes processos podem ser visualizados através da tabela 4.3.

Processo	**Prioridade**	**Período**	***Deadline***	**Tempo de execução**
TA	99	170 us	170 us	20 us
TB	10	170 us	170 us	20 us
TC	9	170 us	170 us	20 us
TD	8	170 us	170 us	20 us
TE	7	170 us	170 us	20 us

Tabela 4.3: Dados dos processos TA, TB, TC, TD e TE.

Este segundo conjunto de processos tem prioridades diferentes para poder mostrar uma sequência lógica de execução, a qual foi induzida de forma que os processos TB, TC, TD e TE dormissem e só acordassem durante a execução do processo TA. Desta forma, o processo TA foi iniciado e pouco tempo depois ocorreu uma interrupção responsável por tratar os temporizadores de alta resolução, que por sua vez foram responsáveis por acordar os outros processos de tempo real (TB, TC, TD e TE). Os valores de tempo das variáveis utilizadas no cálculo das equações (4.1) e (4.2) podem ser verificadas na tabela 4.4. O cálculo para descobrir qual o tempo de interrupção que o processo TA sofreu, aplicando os valores das

variáveis da tabela 4.4, pode ser verificado na equação (4.4).

Variável	Valor
C_{TC}	1397 ns
C_{FG}	1715 ns
$C_{FT}(1)$"TB"	11414 ns
$C_{FT}(2)$"TC"	8027 ns
$C_{FT}(3)$"TD"	13185 ns
$C_{FT}(4)$"TE"	7585 ns

Tabela 4.4: Tempos de execução do segundo conjunto de processos (TA, TB, TC, TD e TE).

$$\begin{aligned} C_{TAR} &= 11414 + 8027 + 13185 + 7585 \\ C_{TAR} &= 40211ns \\ C_T &= 1397 + 1715 + 40211 \\ C_T &= 43323ns \end{aligned} \tag{4.4}$$

Desta forma, o processo TA que deveria processar durante 20000 ns sofreu uma interferência de 43323 ns, ou seja, mais do que o dobro do tempo necessário para terminar o processamento de TA. Os outros processos que foram acordados, atrapalhando assim o processo TA, só puderam ser processadas depois da execução deste. Para visualizar melhor, a figura 4.5 mostra a execução medida no *kernel* deste conjunto de processos.

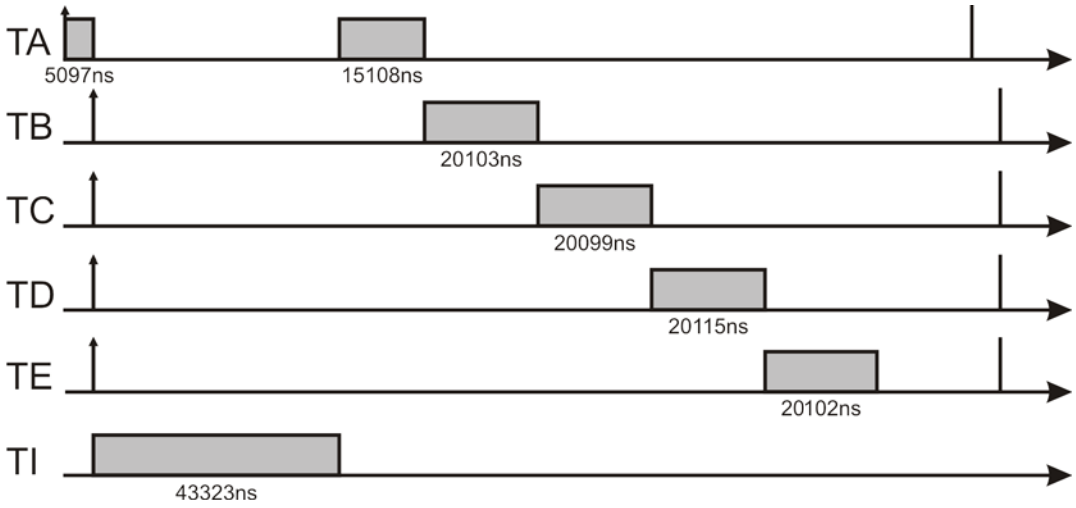

Figura 4.5: Representação da execução dos processos TA, TB, TC, TD e TE.

Semelhante como ocorre na execução da figura 4.4, na execução da figura 4.5 existe a representação do tratador de interrupção (TI), o qual desta vez demora bem mais na

sua execução, pois agora deve acordar 4 processos sequencialmente em vez de apenas um. Assim, o processo TA, acaba sofrendo uma interferência de 43323 nanossegundos para que os processos TB, TC, TD e TE sejam acordados, mesmo que só entrem em execução de fato depois que o processo TA termine de executar.

Isto pode ser visto como uma inversão de prioridades, pois nos dois casos o processo TA, que é a de maior prioridade, é preemptado pelo tratador de interrupção para realizar um processo que diz respeito aos processos de prioridades menores. Desta forma mostra-se um problema que em certas ocasiões pode causar inversão de prioridades, mas em outras pode ser justamente o contrário, fazendo com que os processos de mais alta prioridade sejam postas na fila de prontos durante a execução de processos de menor prioridade, fazendo com que elas iniciem no momento correto, ou mais próximo do correto.

4.3 Comentários

Os temporizadores de alta resolução necessitam de precisão, por causa disto, sua execução não pode ser postergada através de *SoftIRQ*, sendo executados em *hard IRQ*, o que acaba causando interferência a todos os processos em execução, a não ser pelos temporizadores de alta resolução que são marcados para executarem realmente em *SoftIRQ*, já que estes podem ser postergados. Mas todos os temporizadores de alta resolução que são utilizados para acordar um processo são executados em *hard IRQ*, o que leva ao problema encontrado, pois estes temporizadores preemptam todos os processos em execução para acordar outros, só que se algum destes outros não possuírem prioridade maior que a do processo mais prioritário executando na CPU, eles continuarão sem executar, até que possuam a prioridade mais alta entre os processos em execução. Então a inversão de prioridade se encontra justamente na possibilidade de um processo de baixa prioridade ser acordado, interferindo na execução de outros processos com prioridade maior por intermédio dos temporizadores e sua execução em *hard IRQ*.

Para resolver este problema, deve-se postergar a execução dos temporizadores utilizados como *sleeps*, para isto, este trabalho propõe a criação de um processo por CPU, responsável por executar estes temporizadores em momentos posteriores, relacionando suas execuções com as prioridades dos processos que criaram os temporizadores. Assim, o processo proposto herdará a prioridade destes processos a serem acordados, entrando em execução apenas quando o processo de quem ele herdou tivesse que executar. Quando este processo executa, ele acorda o processo de maior prioridade relacionado aos *sleeps* postergados. Sendo assim, quando um processo utiliza um *sleep*, ele causará menos interferência em contexto de interrupção e só acordará realmente quando ele for o processo apto a executar e mais prioritário do sistema, assim ele pode executar no instante que entrar na fila de prontos. A explicação da proposta em mais detalhes pode ser vista no próximo capítulo.

Capítulo 5

Proposta para Reduzir a Interferência do Tratador de Interrupções dos Temporizadores de Alta Resolução

Este trabalho trata sobre Linux de tempo real e sobre possíveis alterações que possam melhorá-lo, justamente em relação ao seu desempenho de tempo real. Desta forma, depois de identificado um problema que diminui este desempenho, foi proposta uma solução. Neste capítulo será explicada detalhadamente esta proposta para resolver o problema descrito no capítulo anterior, a implementação realizada e por fim a análise do Linux utilizando a proposta implementada.

5.1 Proposta

Após analisar uma maneira de diminuir a interferência causada pelo tratador de interrupções dos temporizadores de alta resolução em relação aos processos de tempo real, sem que estes temporizadores percam precisão de forma significante, definiu-se a proposta deste trabalho, a qual é criar uma maneira de postergar o trabalho de acordar um processo, de forma que isto seja executado fora do contexto de interrupção.

Para que isto seja possível, é necessário ter uma forma de postergar o trabalho realizado em contexto de interrupção, fazendo com que seja realizado em contexto de processo, mas mesmo assim, ele ainda deve ser executado em momentos oportunos. A proposta é semelhante a implementação da *softIRQ*, que é uma forma de postergar o trabalho de processos que executariam em contexto de interrupção. Mas difere em relação a como e quando estes

processos devem ser executados, não permitindo que o processo a ser acordado demore muito mais que o necessário para entrar em execução, como também, não podendo atrapalhar os processos de prioridades maiores que a dele.

Este trabalho propõe a criação de uma *thread* de kernel, a qual será responsável por acordar todos os processos que deveriam ser acordados pelos temporizadores de alta resolução em contexto de interrupção, tendo seu trabalho realizado em contexto de processo, acordando o processo apenas quando ele já possa executar, ou seja, sem a necessidade de esperar muito tempo depois que entrar na fila de prontos. Esta *thread* será referenciada neste trabalho por *khrtimer_prio*.

A *thread khrtimer_prio* deve acordar um processo, se e somente se não existir nenhum outro na fila de prontos com prioridade maior que a daquele a ser acordado. Como este trabalho é voltado para Linux de tempo real, a *thread* trabalhará sobre os seus processos de tempo real. Já os processos normais do sistema serão acordados quando não existir nenhum processo de tempo real com prioridade maior que um, a qual é a menor prioridade dos processos de tempo real, pois qualquer processo de tempo real é mais prioritário que os processos normais. Assim, após os processos normais serem acordados, serão executados de acordo com suas prioridades e as regras de sua classe de escalonamento, não interferindo em suas execuções.

Na realidade a thread *khrtimer_prio* irá manipular muito a parte de prioridades do sistema, pois ela deve analisar a prioridade de todos os processos que devem ser acordados e herdar apenas a prioridade do processo mais prioritário entre estes. Desta forma, a thread ao entrar em execução, será considerado o processo mais prioritário no sistema, significando que pode acordar o processo de quem ela herdou a prioridade. Mas para dar a vez de execução definitivamente para este processo, ela herda a prioridade do próximo processo a ser acordado, voltando assim para a fila de prontos. Assim ela sempre dá a certeza que os processos após acordarem entrarão logo em execução.

Os temporizadores de alta resolução são armazenados em uma árvore vermelha e preta, ordenados pelo seu tempo de expiração. Só que para organizar estes temporizadores de forma que a *thread* acorde os processos eficientemente, eles devem ser organizados pela prioridade de seus processos, para assim, a *thread* sempre acordar os que tiverem maior prioridade, de forma mais rápida e eficiente, pois o tempo de busca desta árvore é O(log n). Então para o melhor gerenciamento dos temporizadores expirados, este trabalho também propõe o uso de uma árvore vermelha e preta.

É necessário existir uma *thread khrtimer_prio* por processador, pois cada processador tem seus próprios temporizadores executando localmente. Assim, quando cada processador começar a funcionar, uma *thread* destas deve ser criada para gerenciar os temporizadores utilizados como *sleeps* da CPU específica.

Quando um temporizador responsável por acordar um processo expira, é gerada uma interrupção para tratá-lo, o qual acaba por acordar o processo ligado a este temporizador. Este trabalho propõe alterar isto, de forma que quando a interrupção for gerada, o tratador do *sleep* apenas copiará o temporizador da sua árvore vermelha e preta normal para a árvore de temporizadores expirados, ordenada pelas prioridades dos processos, as quais eles estão relacionados.

Resumindo, com as alterações propostas a *thread khrtimer_prio* herdará a prioridade do processo que possua a mais alta prioridade na árvore de temporizadores expirados. Assim que ela entrar em execução significa que não existe no momento nenhum outro processo com prioridade maior que a dela na fila de prontos. Desta forma, ela pode acordar o processo que solicitou o *sleep*, herdar a prioridade do próximo processo em sua árvore e esperar pela sua próxima execução, liberando o processador para o processo que ele acordou.

Com estas alterações, pretende-se diminuir o tempo gasto no tratador de interrupções correspondente a estes tipos de temporizadores de alta resolução. O que diminui de certa forma a interferência causada por este tipo de inversão de prioridades.

5.2 Implementação da Proposta

Para implementar a proposta deste trabalho é necessário fazer algumas alterações nas estruturas de dados e trechos de código dos temporizadores de alta resolução, como também adicionar partes de código. A seguir é explicado o código da implementação da proposta, começando pelas variáveis necessárias que são declaradas, seguindo pelas funções e estruturas de auxílio da *thread* e por fim a explicação da funcionalidade do código que a *thread* executa.

5.2.1 Variáveis Declaradas

A estrutura *hrtimer*, a qual é fundamental na criação de temporizadores de alta resolução, sofre uma pequena alteração. É adicionado um campo do tipo inteiro, chamado *sleep*, que identificará a utilização dele na atividade de acordar processos, facilitando a identificação destes temporizadores em trechos de código. Este campo é adicionado na estrutura *hrtimer* como mostra a figura 5.1. Sempre que um temporizador for criado para ser utilizado como um *sleep*, esta *flag* deve ser configurada para um valor diferente de zero. Caso o temporizador utilizado como *sleep* seja criado sem o auxílio da função *hrtimer_init_sleeper*, a qual é responsável por iniciar corretamente este temporizador, a *flag sleep* deve ser configurada pelo processo que criou o temporizador, caso contrário este temporizador será tratado como os outros temporizadores de alta resolução. Vale ressaltar que esta identificação dos *sleeps* é feita devido o trabalho ser realizado sobre estes temporizadores, já que os temporizadores de

alta resolução em geral necessitam de precisão e os utilizados como *sleeps* tem a possibilidade de serem postergados, pois eles são utilizados para acordar processos, mas mesmo que um processo seja acordado, se ele não for o processo mais prioritário ele vai ter que esperar por sua execução. Então estes temporizadores podem ser postergados até que os processos que eles devem acordar sejam os mais prioritários da CPU, desta forma, assim que eles acordarem serão executados.

```
<linux/hrtimer.h>

struct hrtimer {
  ...
  int sleep;
};
```

Figura 5.1: Alteração da estrutura *hrtimer*.

Como dito anteriormente neste capítulo, é necessário uma nova árvore vermelha e preta para guardar os temporizadores expirados utilizados como *sleeps*. Para isso, é necessário alterar a estrutura *hrtimer_clock_base*, adicionando a base da nova árvore (*expired_prio*) e um ponteiro para a primeira posição da árvore (*first_prio*), como mostrado na figura 5.2. Estes campos são essenciais para a criação e eficiente manipulação da árvore vermelha e preta.

```
<linux/hrtimer.h>

struct hrtimer_clock_base {
        ...
  struct rb_root     expired_prio;
  struct rb_node     *first_prio;
};
```

Figura 5.2: Alteração da estrutura *hrtimer_clock_base*.

5.2.2 Funções e Estruturas de Auxílio

Os temporizadores criados para funcionarem como *sleeps* geralmente se utilizam das funções *schedule_hrtimeout_range* ou *schedule_hrtimeout*, apresentadas anteriormente no capítulo sobre temporizadores, as quais fazem um processo dormir por um período de tempo determinado. Estas funções utilizam-se da função *hrtimer_init_sleeper* para configurar o temporizador, então ela foi alterada para configurar a variável *sleep* da estrutura *hrtimer*, informando assim que o temporizador trata-se de um *sleep*. A simples alteração pode ser vista na figura 5.3, onde a linha alterada está destacada em negrito.

Como é necessário ordenar os temporizadores pela prioridade dos processos ligados a eles através da estrutura *hrtimer_sleeper*, foi criada uma função para inserir o temporizador na árvore criada e na ordem correta. O algoritmo desta função de inserção do temporizador

```
<kernel/hrtimer.c>

void hrtimer_init_sleeper(struct hrtimer_sleeper *sl, struct task_struct *task)
{
  sl->timer.function = hrtimer_wakeup;
  sl->timer.sleep = 1;
  sl->task = task;
}
```

Figura 5.3: Função *hrtimer_init_sleeper* alterada.

na nova árvore é mostrado na figura 5.4. Ele é escrito na linguagem C, como todo o código escrito para este trabalho.

```
<kernel/hrtimer.c>

#define HRTIMER_STATE_ENQUEUED_SLEEP    0x08

static int enqueue_rbtree_prio(struct hrtimer *timer,
                       struct hrtimer_clock_base *base)
{
  struct rb_node **link = &base->expired_prio.rb_node;
  struct rb_node *parent = NULL;
  struct hrtimer *entry;
  int leftmost = 1;

  while (*link) {
    parent = *link;
    entry = rb_entry(parent, struct hrtimer, node);

    if (hrtimer_get_prio(timer) > hrtimer_get_prio(entry)) {
      link = &(*link)->rb_left;
    } else {
      link = &(*link)->rb_right;
      leftmost = 0;
    }
  }

  if (leftmost)
    base->first_prio = &timer->node;

  rb_link_node(&timer->node, parent, link);
  rb_insert_color(&timer->node, &base->expired_prio);

  timer->state |= HRTIMER_STATE_ENQUEUED_SLEEP;

  return leftmost;
}
```

Figura 5.4: Função que insere um temporizador por prioridade em uma *rbtree*.

Como pode ser observado na figura 5.4, o temporizador é passado por parâmetro através de um ponteiro, juntamente com a base do relógio, a qual é a responsável por identificar em que CPU e em qual base (monotônica ou tempo real) o temporizador deve ser

inserido. O temporizador é analisado com a ajuda de algumas funções criadas para este trabalho e inserido na posição correta, ordenando de forma que o temporizador ligado ao processo com maior prioridade vai ser o primeiro a ser acessado, através do ponteiro *first_prio*. A *flag HRTIMER_STATE_ENQUEUED_SLEEP* é utilizada para informar que o temporizador está inserido na árvore *expired_prio*.

A função de auxílio *hrtimer_get_prio* é utilizada para obter a prioridade do processo ligado ao temporizador, seu algoritmo pode ser visualizado na figura 5.5. Esta função recebe um ponteiro para o temporizador de alta resolução que deve ser avaliado e através da estrutura *hrtimer_sleeper*, a qual ele está ligado, retorna a prioridade do processo que deve ser acordado. Como as prioridades no *kernel* são armazenadas de forma diferente do que é informado quando um processo altera a prioridade de outro, foi criada uma função de auxílio chamada *priority*, que recebe o valor da prioridade de um processo e converte ele para o valor que pode ser utilizado corretamente por algum outro processo.

```
<kernel/hrtimer.c>

static int priority(int prio)
{
  int result = 1;
  if(prio < MAX_RT_PRIO)
    result = (MAX_RT_PRIO - 1) - prio;
  return result;
}

static int hrtimer_get_prio(struct hrtimer *timer)
{
  struct task_struct *t;
  int result = 1;

  t = container_of(timer, struct hrtimer_sleeper, timer)->task;
  if (t) {
    result = priority(t->prio);
  }
  return result;
}
```

Figura 5.5: Funções *priority* e *hrtimer_get_prio*.

É necessário realizar uma pequena alteração na função já existente *__remove_hrtimer*, esta alteração pode ser vista na figura 5.6. A função é responsável por remover o temporizador de sua árvore vermelha e preta. Como foi adicionado uma árvore destas na estrutura dos temporizadores de alta resolução, é necessário alterar a função para poder reconhecer em qual árvore o temporizador está e retirá-lo corretamente.

A parte do código alterado na função mostrada na figura 5.6 está destacada em negrito. A alteração consiste em verificar se o temporizador está com a *flag HRTIMER_STATE_ENQUEUED_SLEEP* ativa, caso sim, significa que ele está na árvore de temporizadores expirados e então pode ser removido da árvore correta.

```
<kernel/hrtimer.c>

static void __remove_hrtimer(struct hrtimer *timer,
                             struct hrtimer_clock_base *base,
                             unsigned long newstate, int reprogram)
{
  if (timer->state & HRTIMER_STATE_ENQUEUED) {
    if (unlikely(!list_empty(&timer->cb_entry))) {
      list_del_init(&timer->cb_entry);
      goto out;
    }
    if(timer->state & HRTIMER_STATE_ENQUEUED_SLEEP) {
      if (base->first_prio == &timer->node) {
        base->first_prio = rb_next(&timer->node);
      }
      rb_erase(&timer->node, &base->expired_prio);
      goto out;
    }
    if (base->first == &timer->node) {
      base->first = rb_next(&timer->node);
      if (reprogram && hrtimer_hres_active(base->cpu_base))
        hrtimer_force_reprogram(base->cpu_base);
    }
    rb_erase(&timer->node, &base->active);
  }
out:
  timer->state = newstate;
}
```

Figura 5.6: Função *__remove_hrtimer* alterada.

Para criar e utilizar a *thread* que vai ser responsável por postergar o trabalho dos *sleeps*, é necessário uma estrutura para referenciar a thread, como também alguns dados extras, então é utilizada uma estrutura de dados chamada *hrtimer_prio_data*, a qual é mostrada na figura 5.7. Esta estrutura possui quatro campos definidos a seguir:

- *cpu*: identifica o processador para qual a *thread* está destinada a funcionar, já que é criada uma *thread* por processador.

- *tsk*: é um ponteiro para a *thread* criada, para assim sempre que necessário referenciar a *thread* corretamente.

- *pending*: é uma *flag* que informa quando a *thread* ainda tem trabalho pendente. Ela possui o valor um quando a *thread* possui temporizadores de *sleeps* expirados na sua árvore, esperando para terem seus processso acordados, caso não tenha trabalho pendente o seu valor é zero.

- *prio*: armazena o valor da prioridade que a *thread* deve herdar, este valor é obtido através do processo com maior prioridade na sua árvore de *sleeps* expirados.

```
<linux/hrtimer.h>

struct hrtimer_prio_data {
  unsigned long        cpu;
  struct task_struct   *tsk;
  int                  pending;
  int                  prio;
};

static DEFINE_PER_CPU(struct hrtimer_prio_data, khrtimer_prio);
```

Figura 5.7: Estrutura *hrtimer_prio_data*.

Definida a estrutura apresentada na figura 5.7 (*hrtimer_prio_data*), é necessário instanciá-la por CPU. Então é utilizada a macro *DEFINE_PER_CPU*, que cria uma instância desta estrutura por processador no sistema, como mostrado também na figura 5.7. Esta macro define internamente um array do tipo da estrutura *hrtimer_prio_data* chamada *khrtimer_prio*, o array possui o tamanho da quantidade de processadores no sistema. Desta forma, a instância por cpu pode ser acessada através de *khrtimer_prio*[CPU], onde CPU é o número do processador responsável pela instância, a qual se quer acessar.

Também é necessário criar uma instância da *thread khrtimer_prio* por processador, isto é feito durante a fase de inicialização do *kernel*. Quando cada processador falha, ou é desligado, esta *thread* também deve parar de executar e ser finalizada. O *kernel* possui alguns métodos para informar quando um processador é iniciado ou finalizado. Para a utilização destes métodos é utilizado um bloco notificador do sistema, o qual deve ser instanciado e configurado para passar as informações para uma função específica. Na figura 5.8 pode ser visto a instanciação do notificador de bloco chamada *hrtimer_prio_cpu_notifier*, a qual é configurada para passar as informações de alterações do estado do processador para a função *hrtimer_prio_cpu_callback*.

A função *hrtimers_prio_init*, apresentada na figura 5.8, é definida através da macro *early_initcall*, para ser executada assim que o *kernel* esteja inicializando. Esta função notifica diretamente à função *hrtimer_prio_cpu_callback* que cada CPU inicializada está funcionando, então a função trata esta informação de acordo com sua programação. Logo após, *hrtimers_prio_init* registra a variável *hrtimer_prio_cpu_notifier* como notificador das alterações de estado da CPU, ou seja, sempre que a CPU sofrer alterações, o sistema será informado através deste notificador, o qual envia a informação para a função *hrtimer_prio_cpu_callback*.

Na figura 5.9 é mostrada a implementação da função *hrtimer_prio_cpu_callback*. Esta função, como visto na figura 5.8, foi registrada para receber as alterações de estado de cada processador e realizar alguma ação de acordo com isto. Esta função deve ter uma assinatura padrão, devendo receber três parâmetros e retornar um valor inteiro. Os parâmetros recebidos devem ser respectivamente um ponteiro do tipo de estrutura *notifier_block*, um valor *unsigned long* que representará a ação e um ponteiro do tipo *void* que deve representar a CPU. As

```
<kernel/hrtimer.c>

static struct notifier_block __cpuinitdata hrtimer_prio_cpu_notifier =
                    { .notifier_call = hrtimer_prio_cpu_callback, };

static __init int hrtimers_prio_init(void)
{
  void *cpu = (void *)(long)smp_processor_id();
  int err = hrtimer_prio_cpu_callback(&hrtimer_prio_cpu_notifer,
                                      CPU_UP_PREPARE, cpu);
  BUG_ON(err == NOTIFY_BAD);
  hrtimer_prio_cpu_callback(&hrtimer_prio_cpu_notifer, CPU_ONLINE, cpu);
  register_cpu_notifier(&hrtimer_prio_cpu_notifer);
  return 0;
}
early_initcall(hrtimers_prio_init);
```

Figura 5.8: Código responsável por notificar o estado do processador.

ações recebidas pela função podem ser:

- *CPU_UP_PREPARE*, *CPU_UP_PREPARE_FROZEN*: estas ações informam que a CPU está se preparando para inicializar, neste momento então, é necessário inicializar os dados de estruturas e variáveis. No trabalho realizado, quando uma destas ações é informada, a função configura a variável *khrtimer_prio* da CPU, especificada por *hcpu*, instancia a *thread khrtimer_prio* e vincula sua execução apenas a CPU informada.
- *CPU_ONLINE*, *CPU_ONLINE_FROZEN*: estas ações informam que a CPU está funcionando e pode começar a executar algo. No caso, quando estas ações são passadas para *hrtimer_prio_cpu_callback*, a função acorda a *thread khrtimer_prio* específica daquela CPU.
- *CPU_UP_CANCELED*, *CPU_UP_CANCELED_FROZEN*, *CPU_DEAD*, *CPU_DEAD_FROZEN*: estas ações informam que a CPU vai parar de funcionar por algum motivo, então deve ser previsto tudo o que pode acabar travando o sistema e fazer algo a respeito, como migrar temporizadores e parar processos. Essas ações apenas são disponíveis quando o *kernel* está configurado para permitir que CPUs possam ser desligadas em tempo de execução. No caso, quando estas ações são passadas para a função, ela finaliza a *thread khrtimer_prio* e chama a função *migrate_hrtimers_prio* daquela CPU, a qual migra os temporizadores para outra CPU ativa. Os processos que estavam sendo processados por aquela CPU também vão ser migrados, assim seus temporizadores devem ser executados, caso contrário o processo não tem como voltar a executar sem ser acordado pelo temporizador.

Na figura 5.9, quando se instancia a *thread khrtimer_prio* através da função *kthread_create*, são passados como parâmetros desta função um ponteiro para a função que a

```
<kernel/hrtimer.c>

static int __cpuinit hrtimer_prio_cpu_callback (struct notifier_block *nfb,
                                        unsigned long action, void *hcpu)
{
  unsigned int cpu = (unsigned long)hcpu;
  struct task_struct *p;
  switch (action) {
  case CPU_UP_PREPARE:
  case CPU_UP_PREPARE_FROZEN:
    per_cpu(khrtimer_prio, cpu).pending = 0;
    per_cpu(khrtimer_prio, cpu).cpu = cpu;
    per_cpu(khrtimer_prio, cpu).tsk = NULL;
    p = kthread_create(khrtimer_prio, &per_cpu(khrtimer_prio, cpu),
                       "khrtimer_prio/%d", cpu);
    if (IS_ERR(p)) {
      printk("khrtimer_prio for %i failed\n", cpu);
      return NOTIFY_BAD;
    }
    kthread_bind(p, cpu);
    per_cpu(khrtimer_prio, cpu).tsk = p;
    break;
  case CPU_ONLINE:
  case CPU_ONLINE_FROZEN:
    wake_up_process(per_cpu(khrtimer_prio, cpu).tsk);
    break;
#ifdef CONFIG_HOTPLUG_CPU
  case CPU_UP_CANCELED:
  case CPU_UP_CANCELED_FROZEN:
  case CPU_DEAD:
  case CPU_DEAD_FROZEN: {
    struct sched_param param;
    param.sched_priority = MAX_RT_PRIO-1;
    p = per_cpu(khrtimer_prio, cpu).tsk;
    sched_setscheduler(p, SCHED_FIFO, &param);
    per_cpu(khrtimer_prio, cpu).tsk = NULL;
    kthread_stop(p);
    migrate_hrtimers_prio(cpu);
    break;
  }
#endif /* CONFIG_HOTPLUG_CPU */
  }
  return NOTIFY_OK;
}
```

Figura 5.9: Código que inicializa e finaliza a estrutura *khrtimer_prio*.

thread executará (*khrtimer_prio*), um ponteiro para a estrutura *khrtimer_prio_data* da CPU a qual a *thread* está ligada, referenciada por *per_cpu(khrtimer_prio, cpu)* e o nome que a *thread* terá no sistema, representada por uma *string* no formato *"khrtimer_prio/%d"*, onde %d será o número da CPU a qual ela estiver ligada, variando de 0 até n-1, onde n é o número de processadores ativos no sistema.

Para o melhor entendimento da função *hrtimer_prio_cpu_callback*, pode-se ver de forma direta o seu funcionamento no fluxograma apresentado na figura 5.10. Lembrando que esta

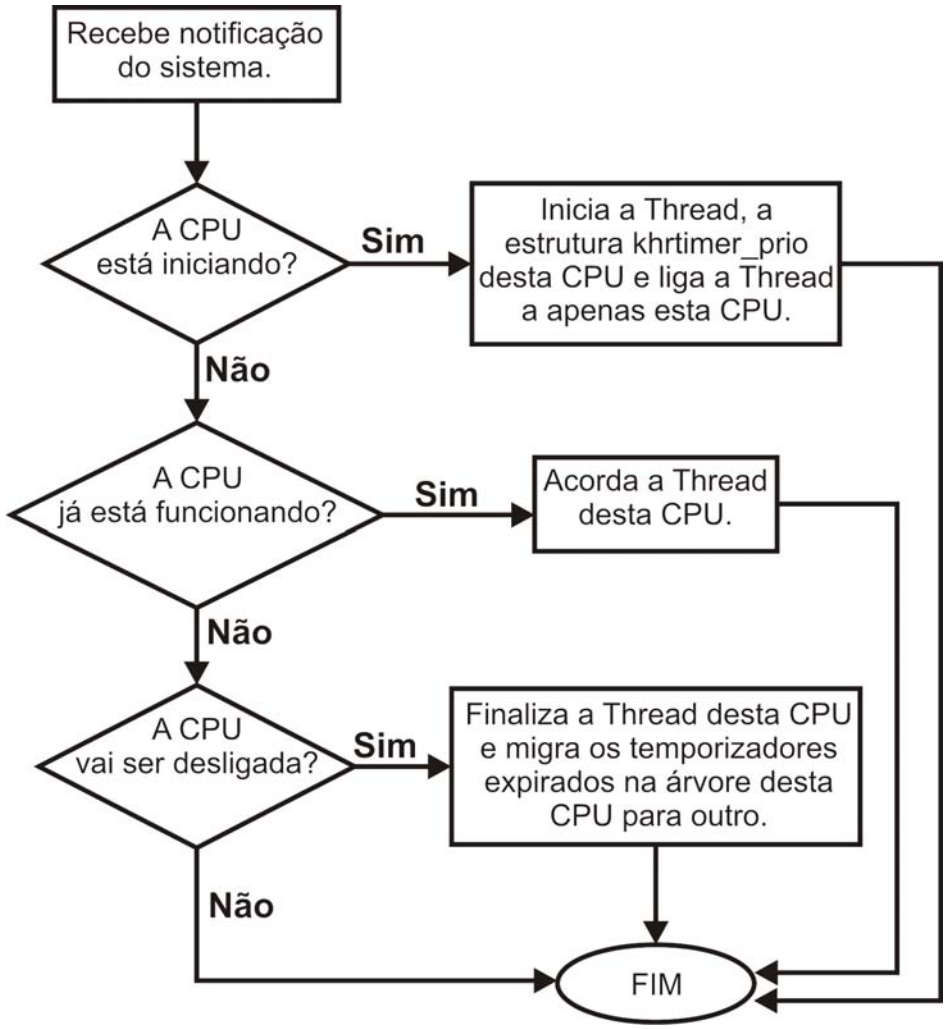

Figura 5.10: Fluxograma do funcionamento da função *hrtimer_prio_cpu_callback*.

função só executa quando chamada diretamente ou quando o sistema muda o estado de uma CPU e tem que informar a ela esta mudança.

O código da função *migrate_hrtimers_prio* é mostrado na figura 5.11. Esta função é utilizada para migrar os temporizadores expirados que estão na árvore de *sleeps* de uma CPU, a qual esteja sendo desativada, para outra CPU que esteja ativa.

A função *migrate_hrtimers_prio* desabilita interrupções locais antes de começar a executar, desta forma não é interrompida no meio do processo enquanto possui alguma variável bloqueada. Depois ela identifica a base dos temporizadores da CPU que está sendo desativada e de uma CPU ativa, para onde serão migrados os temporizadores. É bloqueado o acesso as duas bases, assim não pode haver alterações nelas enquanto é realizado o processo de migração. Todos os temporizadores expirados são removidos de sua árvore e adicionados a árvore de temporizadores expirados da CPU para onde estão sendo migrados. Por fim tudo que teve seu acesso bloqueado é desbloqueado e as interrupções são reativadas. Vale

```
<kernel/hrtimer.c>

static void migrate_hrtimers_prio(int scpu)
{
  struct hrtimer_cpu_base *old_cpu_base, *new_cpu_base;
  struct hrtimer_clock_base *old_base, *new_base;
  struct hrtimer *timer;
  struct rb_node *node;
  int i;

  local_irq_disable();
  old_cpu_base = &per_cpu(hrtimer_bases, scpu);
  new_cpu_base = &__get_cpu_var(hrtimer_bases);
  atomic_spin_lock(&new_cpu_base->lock);
  atomic_spin_lock_nested(&old_cpu_base->lock, SINGLE_DEPTH_NESTING);

  for (i = 0; i < HRTIMER_MAX_CLOCK_BASES; i++) {
    old_base = &old_cpu_base->clock_base[i];
    new_base = &new_cpu_base->clock_base[i];

    while ((node = rb_first(&old_base->active))) {
      timer = rb_entry(node, struct hrtimer, node);

      __remove_hrtimer(timer, old_base, HRTIMER_STATE_MIGRATE, 0);
      timer->base = new_base;
      enqueue_rbtree_prio(timer, new_base);
      timer->state &= ~HRTIMER_STATE_MIGRATE;
    }
  }
  atomic_spin_unlock(&old_cpu_base->lock);
  atomic_spin_unlock(&new_cpu_base->lock);
  local_irq_enable();
}
```

Figura 5.11: Código da função *migrate_hrtimers_prio*.

ressaltar que a execução de uma função destas pode causar um grande overhead, mas ela só é executada se acontecer o desativamento de uma CPU, que é uma ação difícil de ocorrer.

5.2.3 Função da Thread

A *thread* utilizada para postergação do trabalho dos temporizadores, é a principal responsável por diminuir a interferência que os processos sofrem em relação aos temporizadores, pois ela retira o processamento do contexto de interrupção e o executa em outro momento. Como o código que esta *thread* executa é relativamente grande e um pouco complicado de entender, primeiro será apresentado um fluxograma (figura 5.12) que explica seu comportamento de maneira geral. Logo após, o código é explicado em detalhes mais técnicos, mas devido ao seu tamanho ele é dividido em três partes (figuras 5.13, 5.14 e 5.15).

Para o melhor entendimento do fluxograma apresentado na figura 5.12, vale lembrar que a *thread* só executa os temporizadores cujos processos vinculados tenham prioridade

igual a dela, para garantir que só acordará os processos com a maior prioridade no sistema, pois se a *thread* tem a mesma prioridade e está em execução, esta prioridade é a maior entre os processos prontos para executar. Desta forma, a *thread* nem os processos acordados interferem na execução de processos mais prioritários, respeitando assim as prioridades de processos.

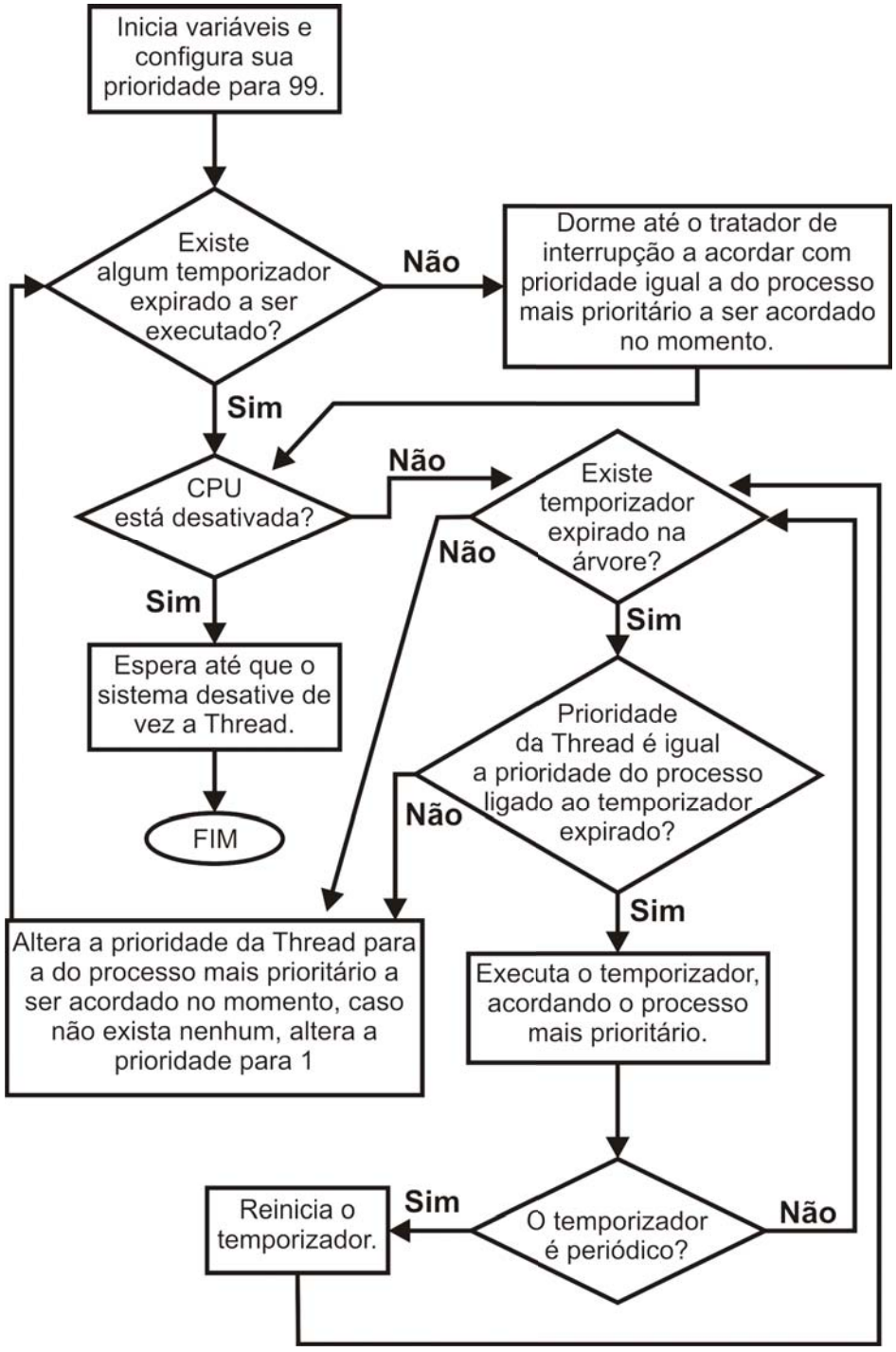

Figura 5.12: Fluxograma do funcionamento da *thread khrtimer_prio*.

Em relação ao código da *thread* em si, na figura 5.13 é apresentado o início do código da função *khrtimer_prio*, onde se apresenta a declaração das variáveis utilizadas. A função recebe como parâmetro um ponteiro, utilizado para passar a estrutura *hrtimer_prio_data* (figura 5.7) da CPU a qual a *thread* se refere. As variáveis declaradas nesta função podem ter sua utilização definida como:

- *param*: utilizada para alterar a prioridade da *thread*.
- *data*: utilizada para realizar a conversão do parâmetro *void *__data* recebido, para a estrutura *hrtimer_prio_data* que é o tipo de estrutura realmente passada.
- *cpu*: informa a CPU a qual esta função está ligada.
- *i*, *prio*, *max_prio*, *pending*, *restart* e *flags*: variáveis de auxílio utilizadas na função.
- *fn*: é um ponteiro para uma função de mesma assinatura que a função de execução do temporizador, utilizada para executar a função do temporizador.
- *cpu_base*, *base*: é um ponteiro para a base de CPU e de relógio respectivamente, utilizada para acessar os temporizadores armazenados.

A *thread* ao iniciar sua execução, altera sua própria prioridade para 99, através do valor da macro *MAX_USER_RT_PRIO* menos um, pois ela possui como valor o número 100. A *thread* se define com a mais alta prioridade do sistema para executar logo o que deve processar em sua inicialização, desta forma, nenhum outro processo de prioridade menor pode atrapalhá-la.

```
<kernel/hrtimer.c>

static int khrtimer_prio(void * __data)
{
  struct sched_param param;
  struct hrtimer_prio_data *data = __data;
  int cpu = data->cpu;
  int i, prio, max_prio, pending, restart;
  enum hrtimer_restart (*fn)(struct hrtimer *);
  struct hrtimer_cpu_base *cpu_base;
  struct hrtimer_clock_base *base;
  unsigned long flags;

  param.sched_priority = MAX_USER_RT_PRIO - 1;
  sys_sched_setscheduler(current->pid, SCHED_FIFO, &param);
  ...
```

Figura 5.13: Parte 1 da função *khrtimer_prio*.

Na figura 5.14 é mostrada a maior parte do código da função *khrtimer_prio*. A execução da *thread* se baseia em um *loop*, assim ela executa seu código repetitivamente até que um

comando do sistema seja enviado, informando que a *thread khrtimer_prio* deve parar de executar. No início de seu *loop* ela verifica se existe algum trabalho pendente a ser feito, caso não exista, ela informa ao processador que vai para a fila de espera e que ele pode escalonar outro processo para execução, ou seja, a thread dorme até que o *kernel* a acorde novamente.

```
<kernel/hrtimer.c>
  ...
  while (!kthread_should_stop()) {
    if (!data->pending) {
      set_current_state(TASK_INTERRUPTIBLE);
      schedule();
    }
    __set_current_state(TASK_RUNNING);
    if (cpu_is_offline(cpu))
      goto wait_to_die;
    local_irq_save(flags);
    cpu_base = &per_cpu(hrtimer_bases,cpu);
    atomic_spin_lock(&cpu_base->lock);
    max_prio = prio = 1;
    pending = 0;
    for (i = 0; i < HRTIMER_MAX_CLOCK_BASES; i++) {
      struct rb_node *node;
      base = &cpu_base->clock_base[i];

      while ((node = base->first_prio)) {
        struct hrtimer *timer;
        timer = rb_entry(node, struct hrtimer, node);
        prio = hrtimer_get_prio(timer);
        if (prio!=priority(current->prio)) {
          pending = 1;
          if (max_prio < prio) max_prio = prio;
          break;
        }
        __remove_hrtimer(timer, base, HRTIMER_STATE_CALLBACK, 0);
        debug_hrtimer_deactivate(timer);
        timer_stats_account_hrtimer(timer);
        fn = timer->function;
        local_bh_disable();
        atomic_spin_unlock(&cpu_base->lock);
        local_irq_restore(flags);
        if (fn)
          restart = fn(timer);
        else
          restart = HRTIMER_NORESTART;
        local_irq_save(flags);
        atomic_spin_lock(&cpu_base->lock);
        _local_bh_enable();
        if (restart != HRTIMER_NORESTART) {
          BUG_ON(timer->state != HRTIMER_STATE_CALLBACK);
          enqueue_hrtimer(timer, base);
        }
        timer->state &= ~HRTIMER_STATE_CALLBACK;
      }//while
    } //for
        ...
```

Figura 5.14: Parte 2 da função *khrtimer_prio*.

Sempre que necessário (como será visto em breve), a *thread khrtimer_prio* é acordada através de um comando do tratador de interrupções dos temporizadores. Quando ela acorda, verifica a CPU, caso esta tenha sido desativada, seu processamento é desviado para um trecho do código responsável por esperar pela finalização correta da *thread*, caso contrário, a *thread khrtimer_prio* continua com sua execução normal. As interrupções daquela CPU são desativadas e a base dos temporizadores de alta resolução da CPU tem seu acesso bloqueado.

Enquanto existir temporizadores expirados na árvore vermelha e preta, o algoritmo pegará o de maior prioridade e verificará se ele possui a mesma prioridade que a *thread khrtimer_prio*, caso não seja igual, significa que a *thread* deve ter sua prioridade alterada. Antes de alterar a prioridade, a *thread* avisa através de uma *flag*, que existe trabalho pendente, desbloqueia a base dos temporizadores e reativa as interrupções locais, pois assim que for alterada a sua prioridade, ela pode ter que sair da fila de execução do processador e ir para a fila de prontos. Caso exista algum processo de prioridade maior que a dela, ela é obrigada a esperar por sua execução. Quando ela puder executar e sua prioridade for igual a prioridade do processo ligado ao temporizador, este tem sua função processada, ou seja, acorda o processo ligado a ele. Caso o temporizador esteja configurado para ser periódico, ele é posto de volta na árvore de temporizadores ativos para cumprir mais um período igual ao anterior.

```
<kernel/hrtimer.c>

    ...
    atomic_spin_unlock_irqrestore(&cpu_base->lock, flags);

    data->pending = pending;
    param.sched_priority = max_prio;
    sys_sched_setscheduler(current->pid, SCHED_FIFO, &param);
  }//while (!kthread_should_stop())
  __set_current_state(TASK_RUNNING);
  return 0;

wait_to_die:
  preempt_enable();
  set_current_state(TASK_INTERRUPTIBLE);
  while (!kthread_should_stop()) {
    schedule();
    set_current_state(TASK_INTERRUPTIBLE);
  }
  __set_current_state(TASK_RUNNING);
  return 0;
}
```

Figura 5.15: Parte 3 da função *khrtimer_prio*.

Quando não existir mais temporizadores expirados na árvore, a *thread* desbloqueia a base dos temporizadores de alta resolução da CPU, reativa as interrupções locais e volta a dormir, até ser acordada por algum evento, como mostra o código das figuras 5.14 e 5.15.

Na figura 5.15, ainda é mostrado o trecho de código que é responsável por fazer a *thread* ser desativada. Geralmente esse código é executado quando o processador vai ser desativado por algum motivo, então a *thread* pára sua execução e muda para a fila de espera do processador, depois o sistema desativa a *thread* definitivamente. Este trecho evita que a thread fique no estado zumbi, como explicado anteriormente.

```
<kernel/hrtimer.c>

void hrtimer_interrupt(struct clock_event_device *dev)
{
  ...
  int raise_prio = 0, temp, prio = 1;
  ...
  for (i = 0; i < HRTIMER_MAX_CLOCK_BASES; i++) {
    ...
    while ((node = base->first)) {
      ...
      temp = hrtimer_rt_defer_prio(timer);
      if (temp == 2) {
        raise_prio = 1;
        if (prio < hrtimer_get_prio(timer))
          prio = hrtimer_get_prio(timer);
      }
      else if(temp == 1)
        raise = 1;
    }
    base++;
  }
  ...
  if (raise_prio)
    wake_up_prio(prio);

  if (raise)
    raise_softirq_irqoff(HRTIMER_SOFTIRQ);
}
```

Figura 5.16: Alterações do código do tratador de interrupções dos *hrtimers*.

Sempre que um temporizador de alta resolução expira, uma interrupção é gerada e deve ser tratada, para isso é executado o tratador de interrupções destes temporizadores (*hrtimer_interrupt*). Para fazer com que os temporizadores utilizados como *sleeps* sejam processados através da *thread* criada, é necessário alterar um trecho do código deste tratador de interrupções como é mostrado na figura 5.16 (as alterações realizadas estão destacadas em negrito). Neste código são adicionadas três variáveis auxiliares e alterada a chamada a função *hrtimer_rt_defer* para *hrtimer_rt_defer_prio*. Quando *raise_prio* possuir valor diferente de zero, ou seja, o temporizador é utilizado como *sleep* e deve ser tratado pela *thread* criada neste trabalho, então é chamada a função *wake_up_prio*.

O código das funções *hrtimer_rt_defer_prio* e *wake_up_prio* é mostrado na figura 5.17. A função *hrtimer_rt_defer_prio* verifica se o temporizador deve ser executado em contexto de interrupção, através da *thread* criada para este trabalho, ou se ele pode ser postergado através

de *softIRQ* e retorna zero, dois ou um respectivamente. A função *wake_up_prio* é responsável por acordar a *thread*, a qual irá processar os temporizadores e mudar a sua própria prioridade para a do temporizador de mais alta prioridade processado pelo tratador de interrupções, mas somente se a prioridade da *thread* for menor que esta.

```
<kernel/hrtimer.c>

static int hrtimer_rt_defer_prio(struct hrtimer *timer)
{
  if(timer->irqsafe) {
    __run_hrtimer(timer);
    return 0;
  }
  __remove_hrtimer(timer, timer->base, timer->state, 0);

  if(timer->sleep) {
    enqueue_rbtree_prio(timer, timer->base);
    if (__get_cpu_var(khrtimer_prio).prio < hrtimer_get_prio(timer)) {
      __get_cpu_var(khrtimer_prio).prio = hrtimer_get_prio(timer);
    }
    return 2;
  }
  list_add_tail(&timer->cb_entry, &timer->base->expired);
  return 1;
}

static void wake_up_prio(int prio)
{
  struct task_struct *tsk = __get_cpu_var(khrtimer_prio).tsk;
  struct sched_param param = { .sched_priority = prio};

  if (tsk && tsk->state != TASK_RUNNING)
    wake_up_process(tsk);
  if (__get_cpu_var(khrtimer_prio).prio < prio) {
    sys_sched_setscheduler(tsk->pid, SCHED_FIFO, &param);
    __get_cpu_var(khrtimer_prio).prio = prio;
  }
}
```

Figura 5.17: Código das funções *hrtimer_rt_defer_prio* e *wake_up_prio*.

De forma geral, de acordo com a implementação desta proposta, sempre que um temporizador utilizado como *sleep* expirar, ele será transferido para uma árvore vermelha e preta de temporizadores expirados e a *thread khrtimer_prio* será informada disto. A *thread* terá sua prioridade alterada para a mais alta entre os processos ligados aos temporizadores expirados, podendo então executar e acordar o processo ligado ao temporizador, ou ficar na fila de prontos esperando por sua execução, caso haja algum processo com prioridade maior. Após executar o processo do temporizador, este é excluído da árvore e a *thread* altera sua prioridade para a prioridade do processo ligado ao próximo temporizador da árvore. Desta maneira a *thread* executa todos os temporizadores expirados, até que a árvore de temporizadores expirados não tenha mais nenhum temporizador, então a *thread* dorme e espera ser acordada pelo tratador de interrupções dos temporizadores quando houver mais processos a

serem acordados pelos temporizadores.

5.3 Medições

Com as alterações realizadas no *kernel* do Linux, é necessário desenvolver uma nova equação para medir o tempo de interferência causado pelo tratador de interrupções dos temporizadores de alta resolução, juntamente com o novo tratamento dos temporizadores utilizados como *sleeps*, realizado através da *thread* criada neste trabalho. Também é necessário realizar novas medições para avaliar a validade desta equação, cujo o cálculo pode ser realizado através da equação (5.1).

$$\begin{aligned} C_T &= C_{TC} + C_{FG} + C_{TAR} \\ C_{TAR} &= C_{ATH} + \sum_{i=1}^{n} \Big(C_{TA}(i) \Big) \end{aligned} \tag{5.1}$$

As variáveis da equação (5.1) podem ser descritas como:

- $C_{TA}(i)$: É o tempo que o tratador de interrupções dos temporizadores de alta resolução modificado gasta para trocar o temporizador i de árvore. Retirando ele da sua árvore vermelha e preta normal e inserindo na árvore que deve armazenar estes temporizadores utilizados como *sleeps* e que tenham expirado (*expired_prio*), ordenando-os pela prioridade do processo ligado a eles.
- C_{ATH}: É o tempo que o tratador de interrupções alterado gasta para poder acordar a *thread* criada neste trabalho. Este tempo pode variar devido o estado da *thread*, já que ela pode estar na fila de espera ou na fila de prontos.
- C_{TAR}: É o somatório do tempo gasto para mudar de árvore todos os temporizadores expirados e utilizados como *sleeps*, mais o tempo gasto por acordar a *thread* criada para este trabalho.
- C_{FG}: Tempo gasto para o tratador de interrupções processar todas as funções gerais da sua rotina, como atualizar estatísticas, variáveis entre outras tarefas;
- C_{TC}: Tempo gasto nas trocas de contexto entre o processo que estava executando e o código do tratador de interrupções;
- C_T: Tempo de execução gasto pelo tratador de interrupções para processar todos os temporizadores de alta resolução expirados;

A equação (5.1) utiliza o tempo de troca de árvore dos temporizadores, em vez do tempo de acordar realmente o processo ligado ao temporizador, como é feito no kernel padrão estudado, o que faz com que desta forma diminua o tempo gasto pelo tratador de interrupções. O tempo que o tratador de interrupções gasta para trocar um temporizador de árvore pode ser analisado no gráfico da figura 5.18, o qual está na mesma escala do gráfico da figura 4.3 que é repetido na figura 5.19 para facilitar a comparação.

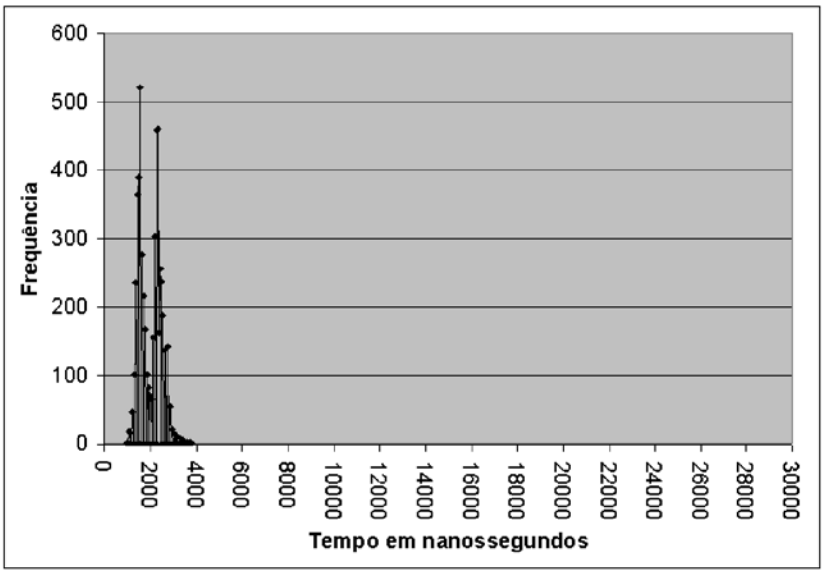

Figura 5.18: Tempo gasto para trocar um temporizador de árvore.

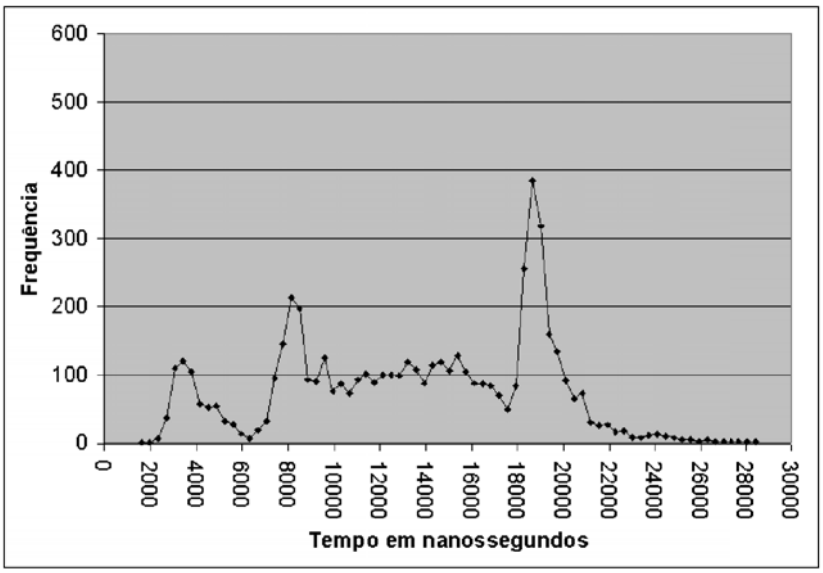

Figura 5.19: Variação de tempo para acordar um processo através do tratador de interrupções.

Comparando o tempo de acordar um processo e de trocar um temporizador de árvore, percebe-se que acordar um processo na maioria das vezes tem um custo muito alto em relação à troca de árvore do temporizador. Desta forma, com as alterações realizadas no tratador de interrupções do *kernel* do Linux, o tempo gasto por ele diminui em relação aos temporizadores

utilizados como *sleeps*. Para analisar melhor como se comporta o *kernel* do Linux com as alterações realizadas, foi utilizado primeiramente o conjunto de processos (TA e TB) apresentado no capítulo 4 na tabela 4.1. Através de medições realizadas diretamente no *kernel* alterado, pôde-se obter os tempos de execução necessários para realizar o cálculo de quanto tempo o tratador de interrupções dos temporizadores interferiram na execução dos processos, os tempos utilizados para este cálculo podem ser verificados na tabela 5.1.

Variável	**Valor**
C_{TC}	1612 ns
C_{FG}	1814 ns
$C_{TA}(1)$"TB"	2148 ns
C_{ATH}	2708 ns

Tabela 5.1: Tempos do conjunto de processos TA e TB (*kernel* alterado).

Aplicando os valores da tabela 5.1 à equação (5.1) obtemos o tempo total que o tratador de interrupções dos temporizadores de alta resolução gastou na interferência da execução do processo TA. Os cálculos deste conjunto de processos podem ser vistos na equação (5.2). O tempo total de interferência causada pelo tratador de interrupções com as alterações propostas no *kernel* do Linux é de 8282 nanossegundos, enquanto que o tempo total desta interferência no *kernel* do Linux estudado é de 15376 nanossegundos. Neste caso, houve uma diminuição de aproximadamente 46% do tempo gasto na interferência causada pelo tratador de interrupções.

$$\begin{aligned} C_{TAR} &= 2708 + 2148 = 4856ns \\ C_T &= 1612 + 1814 + 4856 = 8282ns \end{aligned} \tag{5.2}$$

Para o melhor entendimento e fácil visualização de como ocorreu a execução deste conjunto de processos no *kernel* do Linux alterado, pode-se analisar a figura 5.20, a qual representa graficamente a linha de tempo desta execução. Para facilitar ainda a comparação entre a execução do *kernel* normal e do *kernel* alterado, a figura (4.4) é repetida, representada pela figura 5.21.

Analisando a figura 5.20, nota-se que o processo TA inicia sua execução, pouco tempo depois TB deve acordar, devido a isso, seu temporizador gera uma interrupção, o que faz com que o tratador de interrupções (TI) entre em ação. Com o código alterado, TI irá apenas trocar de árvore o temporizador que deve acordar TB e acordar a *thread* (TX), mudando sua prioridade para a mesma que a do processo TB. Desta forma, TI não executa por um tempo maior que o necessário. Quando TI termina de executar, o processo TA consegue continuar e

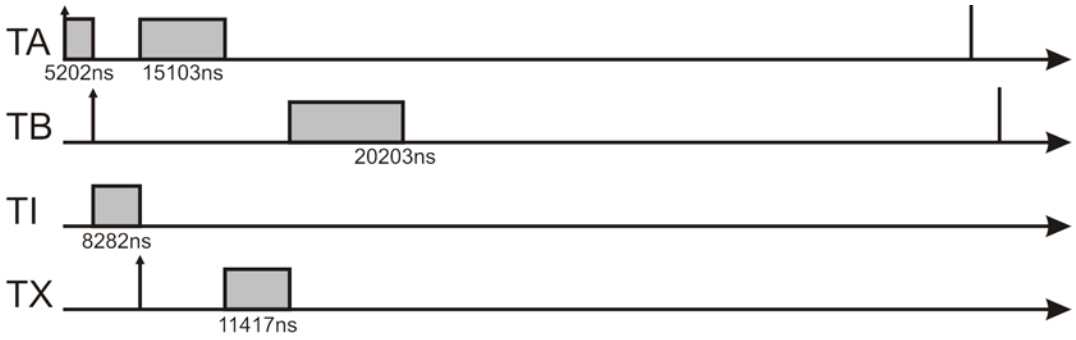

Figura 5.20: Representação da execução dos processos TA e TB (*kernel* alterado).

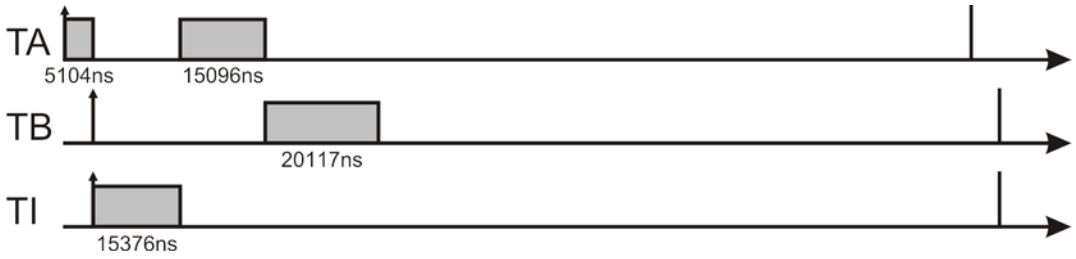

Figura 5.21: Representação da execução dos processos TA e TB (*kernel* normal).

concluir sua execução, saindo assim da fila de prontos. Então o processo TX que no momento é o processo com maior prioridade na fila de prontos, executa e acorda o processo TB que está ligado ao temporizador. Quando TX termina de executar, volta a dormir e TB que agora é o processo de mais alta prioridade na fila de prontos, pode finalmente executar.

Comparando a figura 5.20 com a figura 5.21, pode-se verificar que o processo mais prioritário (TA) sofre uma interferência significantemente menor. Continuando a análise, nota-se que o processo (TB) que é acordado pelo temporizador, executa em um tempo um pouco posterior que na execução com o *kernel* sem alterações. Isto acontece por causa da inclusão de um novo processo (TX) neste meio, aumentando um pouco o tempo de início de execução dos processos que devem ser acordados, já que o processo TX gasta tempo com troca de contexto a cada vez que ele deve acordar um processo.

Para mostrar o funcionamento do *kernel* alterado com um exemplo um pouco maior e mais complexo, foi utilizado o segundo conjunto de processos utilizado no capítulo 4 (TA, TB, TC, TD e TE), o qual teve seus tempos de execução medidos no kernel e são apresentados na tabela 5.2.

Aplicando os valores da tabela 5.2 à equação (5.1), obtemos o tempo total que o tratador de interrupções gasta para mudar os temporizadores de TB, TC, TD e TE para outra árvore. Os cálculos deste conjunto de processos podem ser vistos na equação (5.3).

Variável	Valor
C_{TC}	1514 ns
C_{FG}	1693 ns
$C_{TA}(1)$"TB"	1687 ns
$C_{TA}(2)$"TC"	2322 ns
$C_{TA}(3)$"TD"	2447 ns
$C_{TA}(4)$"TE"	2335 ns
C_{ATH}	2404 ns

Tabela 5.2: Tempos do conjunto de processos TA, TB, TC, TD e TE (*kernel* alterado).

$$\begin{aligned} C_{TAR} &= 2404 + 1687 + 2322 + 2447 + 2335 = 11195ns \\ C_T &= 1514 + 1693 + 11195 = 14402ns \end{aligned} \tag{5.3}$$

Segundo o cálculo da equação (5.3) sobre o conjunto de processos TA, TB, TC, TD e TE, o tempo total de interferência causado foi de 14402 nanossegundos no *kernel* alterado. Esta mesma medição no *kernel* sem alterações foi de 43323 nanossegundos, obtendo aproximadamente 67% de redução do tempo gasto com esta interferência. Para visualizar e entender melhor como ocorreu esta execução no *kernel* do Linux alterado e poder comparar o mesmo conjunto de processos no *kernel* normal, pode-se analisar as figuras 5.22 e 5.23, as quais representam as execuções deste conjunto de processos no *kernel* alterado e no normal respectivamente.

Analisando a figura 5.22, pode-se perceber que o processo TA inicia sua execução antes de qualquer outro processo, pouco tempo depois os processos TB, TC, TD e TE deveriam acordar, o que faz com que o tratador de interrupções (TI) entre em execução para tratar os temporizadores ligados a estes processos. Então, TI transfere os quatro temporizadores responsáveis por acordar estes processos para a árvore de temporizadores utilizados como *sleeps*, logo em seguida acorda a *thread* (TX) e muda a prioridade dela para a prioridade mais alta de todos estes processos que serão acordados, que no caso é a prioridade do processo TB. Quando TI termina de executar e o processo TA termina sua execução, TA sai da fila de prontos, então TX agora possui a prioridade mais alta entre os processos na fila de prontos, o que o faz entrar em execução. TX acorda o processo TB e muda sua própria prioridade para a do próximo processo que deve ser acordado (o processo TC). Quando a prioridade de TX é alterada, o escalonador verifica que no momento o processo TB possui a maior prioridade entre os processos na fila de prontos e faz com que o processo TX desocupe o processador e TB seja executado. Isto ocorre recursivamente, pois quando TB termina de executar, ele sai da fila de prontos, então TX volta a ser o processo de mais alta prioridade na fila de prontos e executa acordando o processo TC, logo após muda sua própria prioridade para a mesma

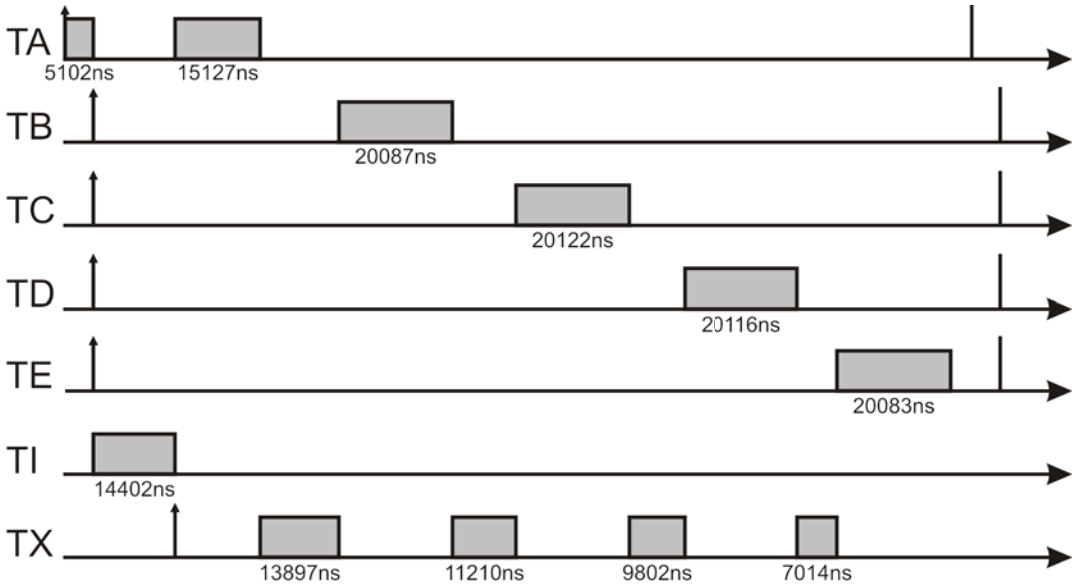

Figura 5.22: Representação da execução dos processos TA, TB, TC, TD e TE (kernel alterado).

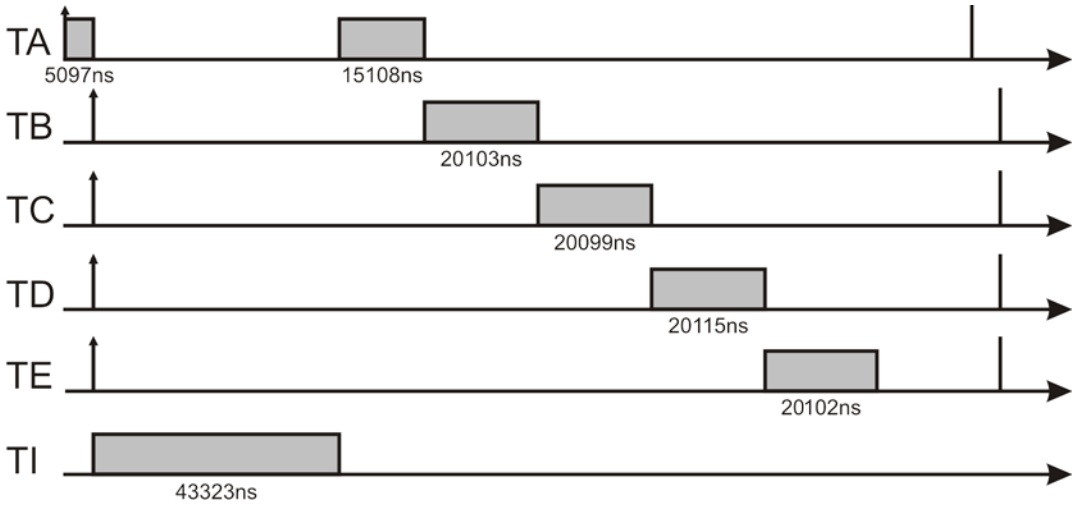

Figura 5.23: Representação da execução dos processos TA, TB, TC, TD e TE (kernel normal).

que o processo TD possui. Assim ocorre até que TX acorde o processo TE e volte a dormir, terminando este ciclo de execução com a conclusão do processo TE.

Comparando esta execução no *kernel* do Linux sem alterações (apresentado na figura 4.5 e repetida aqui pela figura 5.23 para facilitar a comparação) e no *kernel* alterado (apresentado na figura 5.22), percebe-se algo semelhante à comparação anterior, onde o processo mais prioritário é executado em um tempo significantemente menor. Só que entre os demais processos que devem ser acordados, os menos prioritários acabam sofrendo um atraso maior, pois na comparação pode-se notar que os processos TB e TC (mais prioritários) têm suas execuções iniciadas no *kernel* alterado em um tempo menor que quando iniciados no *kernel* sem alterações. Já os processos TD e TE (menos prioritários) têm suas execuções iniciadas no *kernel* alterado um pouco depois que quando iniciados no *kernel* normal. Isto ocorre pelo

mesmo motivo já mencionado, pois existe um novo processo (TX) neste meio que gera este atraso, o qual cresce levemente a cada processo que deve ser acordado em sequência, pois é o somatório do tempo que ele gasta a cada troca de contexto. Vale ressaltar ainda que, os processos mais prioritários são executados antes no *kernel* alterado do que no outro, devido a diminuição considerável do tempo gasto na interferência do tratador de interrupções.

5.4 Comentários

Com as alterações realizadas no *kernel* do Linux, conseguiu-se postergar o trabalho do tratador de interrupções relacionado aos temporizadores de alta resolução. A postergação deste trabalho é realizada por partes, deixando de executar de uma só vez todos os temporizadores utilizados como *sleeps*, para executá-los um a um pouco antes da execução do processo que eles devem acordar. Desta forma, a interferência identificada por este trabalho é reduzida de forma significante, diminuindo o tempo que o processo de alta precisão em execução deve esperar para que o sistema trate a interrupção causada por estes temporizadores, como também diminui o tempo do início da execução dos processos mais prioritários a serem acordados. Mas para melhorar a situação da execução dos processos de mais alta prioridade em relação aos temporizadores, acabou sendo necessário aumentar, mesmo que as vezes de forma insignificante, o tempo que alguns processos menos prioritários devem esperar para entrarem na fila de prontos e assim começarem a executar. Ainda assim, vale ressaltar por fim, que a inversão de prioridade existente no *kernel* estudado e identificado neste trabalho, foi resolvida com esta postergação de trabalho, pois de acordo com a proposta da resolução do problema desta dissertação, os temporizadores criados pelos processos herdam de certa forma suas prioridades e respeitam assim a ordem de execução dos processos, a qual é de acordo com essas prioridades.

Capítulo 6

Conclusão

Temporizadores de alta resolução são muito utilizados no Linux por possuírem alta precisão, mas para garantir esta precisão eles geram uma interrupção assim que expiram, a qual é tratada pelo seu tratador de interrupções, que por sua vez preempta qualquer processo em execução para processar a função ligada ao temporizador. Desta forma, estes temporizadores interferem na execução de qualquer processo, sendo ele de alta ou baixa prioridade. Só que alguns destes temporizadores, dependendo das suas funções, não necessitam de uma precisão tão alta, nem podem ser executados com precisão muito baixa, sendo executados um pouco depois de expirarem sem prejuízo e sem causar tanta interferência ao processo que estiver em execução na CPU. Mas existem temporizadores utilizados para acordar processos em determinados momentos, conhecidos geralmente como *sleeps*, que executam com alta precisão, mas que na verdade podem executar em momentos posteriores, pois um processo, dependendo da sua classe de escalonamento, só pode executar depois que ele for o processo mais prioritário na CPU. Então mesmo que ele acorde com alta precisão, se houver algum processo com prioridade maior, o processo acordado além de ter interferido na execução deste, ainda não vai poder executar.

O problema identificado se encontra no ato de processos de baixa prioridade serem acordados durante a execução de processos de alta prioridade, interferindo nesta execução e criando uma inversão de prioridades, já que os processos de alta prioridade estão deixando de executar para esperar que processos de baixa prioridade sejam acordados.

Para resolver o problema foi criada uma *thread* que posterga a execução das funções ligadas aos temporizadores utilizados como *sleeps*. Desta forma, cada temporizador destes só acorda o processo vinculado a ele quando a prioridade do processo for a maior da CPU, significando assim que, quando eles forem acordados entrarão em execução em seguida. Assim, além de reduzir a interferência causada aos processos que estejam em execução no momento da interrupção, resolve o problema de inversão de prioridades, pois os processos de menor prioridade não vão mais ser acordados durante a execução dos processos de maior prioridade.

A proposta para resolver o problema foi implementada no *kernel* do Linux e puderam-se realizar medições para validar seus objetivos. Através destas medições constatou-se que com a postergação do trabalho do tratador de interrupções em relação aos *sleeps*, a interferência ocorrida sobre o processo que estiver em execução diminui de forma que, quanto mais temporizadores expirados ao mesmo tempo forem postergados, maior é a diferença entre a execução normal deste tratador de interrupções e sua execução de acordo com a proposta implementada. Quando um temporizador destes é postergado, chega-se a reduzir em média cerca de nove vezes o tempo que ele gasta executando em contexto de interrupção. Constatou-se também que com a diminuição deste tempo de interferência, além do processo que é interrompido conseguir terminar sua execução em menor tempo, dependendo da quantidade de temporizadores postergados, também aumenta o número de processos de alta prioridade que terminam de executar em menor tempo. Só que da mesma forma que os processos de maior prioridade terminam em menor tempo, dependendo também da quantidade de temporizadores postergados, os processos a serem acordados e com menor prioridade ou que são acordados por último terminam sua execução em um tempo maior, tempo este incrementado pelo processamento da *thread* executando cada temporizador expirado. Além destas diferenças nos tempos de execução dos processos de tempo real do Linux, a proposta consegue resolver o problema de inversão de prioridades, o qual é um problema não aceitável em relação a sistemas de tempo real.

Com as alterações propostas, o problema encontrado é resolvido, mas o *overhead* é incrementado, já que somando o tempo gasto em contexto de interrupção com o processamento realizado para postergar o trabalho e executar os temporizadores em contexto de processo, o tempo estimado para se executar estes temporizadores aumenta.

O protótipo implementado funciona com vários processadores executando em paralelo, já que as estruturas utilizadas foram instanciadas por processador, assim cada CPU possui seus dados separados. Como as interrupções geradas pelos temporizadores são por CPU, cada uma executa suas interrupções independente da outra, desta forma, foi instanciada uma *thread* para realizar a postergação do trabalho do tratador de interrupções por CPU, assim cada *thread* instanciada é ligada a uma única CPU, não podendo executar por outra CPU que não seja a qual ela estiver ligada.

Para a implementação desta proposta no *kernel* do Linux, foram deletadas apenas três linhas do seu código normal e modificadas mais duas linhas, mas foram adicionadas um total de 204 linhas de código, o que torna a manutenção deste código para modificações ou versões futuras relativamente simples. Todo o código está apresentado e explicado neste trabalho, ele é relativamente pequeno e mesmo que as alterações realizadas mudem o fluxo de execução de uma pequena parte do *kernel*, as linhas alteradas de seu código são mínimas, na sua grande maioria o trabalho adiciona um novo código para manipular esta mudança do fluxo de execução, o qual está detalhado nesta dissertação, desde o seu funcionamento até o seu código.

Em relação a trabalhos futuros, pode-se pensar nas seguintes abordagens:

- Estudo de outros meios de postergar os temporizadores de alta resolução.
- Estudar a viabilidade dos temporizadores herdarem as prioridades dos processos que os criaram, para assim, quando mais que um temporizador expirar ao mesmo tempo, eles possam ser executados de acordo com a importância destes processos.
- Estudar a viabilidade de desabilitar a interrupção dos temporizadores ligados a processos menos prioritários que o processo em execução. Diminuindo o tempo de interferência que esta interrupção causaria.

Apêndice A

Macros para comparar *ticks*

Para comparar *ticks* corretamente, o *kernel* do Linux define as seguintes macros:

- *time_after(a, b)* e *time_after64(a, b)*: Estas duas macros retornam verdadeiro se "a"ocorre depois de "b", se não retornam falso, sendo a primeira macro utilizada para variáveis de 32 *bits* e a segunda para variáveis de 64 *bits*.
- *time_before(a, b)* e *time_before64(a, b)*: Retornam verdadeiro se "a"ocorre antes de "b", se não retornam falso, sendo a primeira macro utilizada para variáveis de 32 *bits* e a segunda para variáveis de 64 *bits*.
- *time_after_eq(a, b)* e *time_after_eq64(a, b)*: Retornam verdadeiro se "a"ocorre depois ou no mesmo instante de "b", se não retornam falso, como as demais a segunda é utilizada para variáveis de 64 *bits*.
- *time_before_eq(a, b)* e *time_before_eq64(a, b)*: Retornam verdadeiro se "a"ocorre antes ou no mesmo instante de "b", se não retornam falso, como as demais a segunda é utilizada para variáveis de 64 *bits*.
- *time_in_range(a, b, c)*: Retorna verdadeiro se "a"ocorre depois ou no mesmo instante de "b"e antes ou no mesmo instante de "c", se não retorna falso.
- *time_is_after_jiffies(a)*: Retorna verdadeiro se "a"ocorre depois de *jiffies*, caso contrário retorna falso.
- *time_is_before_jiffies(a)*: Retorna verdadeiro se "a"ocorre antes de *jiffies*, caso contrário retorna falso.
- *time_is_after_eq_jiffies(a)*: Retorna verdadeiro se "a"ocorre depois ou no mesmo instante de *jiffies*, caso contrário retorna falso.

- *time_is_before_eq_jiffies(a)*: Retorna verdadeiro se "a"ocorre antes ou no mesmo instante de *jiffies*, caso contrário retorna falso.

Referências Bibliográficas

[1] Rafael Vidal Aroca. *Análise de sistemas operacionais de tempo real para aplicações de robótica e automação*. Dissertação (mestrado), Escola de Engenharia de São Carlos - Universidade de São Paulo, São Paulo, 2008.

[2] Siro Arthur, Carsten Emde, and Nicholas Mc Guire. Assessment of the realtime preemption patches (rt-preempt) and their impact on the general purpose performance of the system. In *Proceedings of the 9th Real-Time Linux Workshop*, 2007.

[3] Arun R. Bharadwaj. *Timers: Framework for identifying pinned timers*. Disponível em: http://lwn.net/Articles/327516/ - Último acesso em: 12 jan 2010, 2009.

[4] Carlos Boneti, Roberto Gioiosa, Francisco J. Cazorla, and Mateo Valero. A dynamic scheduler for balancing hpc applications. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008.

[5] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2000.

[6] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. 3 ed. Disponível em: http://book.opensourceproject.org.cn/kernel/kernel3rd/ - Último acesso em: 01 maio 2009, 2005.

[7] Randy Brown. Calendar queues: A fast o(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, pages 1220–1227, 1988.

[8] Andreu Carminati and Rômulo Silva de Oliveira. Interferência das hard irqs e softirqs em tarefas com prioridade de tempo real no linux. *Workshop de Sistemas Operacionais*, 2009.

[9] Jonathan Corbet. *Deleting timers quickly*. Disponível em: http://lwn.net/Articles/84836/ - Último acesso em: 10 abr 2009, 2004.

[10] Jonathan Corbet. *The dynamic tick patch*. Disponível em: http://lwn.net/Articles/138969/ - Último acesso em: 13 abr 2009, 2005.

[11] Jonathan Corbet. *A new approach to kernel timers*. Disponível em: http://lwn.net/Articles/152436/ - Último acesso em: 18 mar 2009, 2005.

[12] Jonathan Corbet. *How fast should HZ be?* Disponível em: http://lwn.net/Articles/145973/ - Último acesso em: 02 abr 2009, 2005.

[13] Jonathan Corbet. *The high-resolution timer API.* Disponível em: http://lwn.net/Articles/167897/ - Último acesso em: 17 abr 2009, 2006.

[14] Jonathan Corbet. *Clockevents and dyntick.* Disponível em: http://lwn.net/Articles/223185/ - Último acesso em: 27 abr 2009, 2007.

[15] Jonathan Corbet. *Deferrable timers.* Disponível em: http://lwn.net/Articles/228143/ - Último acesso em: 23 abr 2009, 2007.

[16] Jonathan Corbet. *High- (but not too high-) resolution timeouts.* Disponível em: http://lwn.net/Articles/296578/ - Último acesso em: 19 abr 2009, 2008.

[17] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms.* The MIT press, 2000.

[18] Lorenzo Dozio and Paolo Mantegazza. Linux real time application interface (rtai) in low cost high performance motion control. In *Proceedings of the conference of ANIPLA, Associazione Nazionale Italiana per l'Automazione*, 2003.

[19] Morten Engen. *Better Real-Time Capabilities For The AVR32 Linux Kernel.* PhD thesis, Institutt for teknisk kybernetikk. Norwegian University of Science, 2007.

[20] K. Bruce Erickson, Richard E. Ladner, and Anthony LaMarca. Optimizing static calendar queues. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, pages 179–214, 2000.

[21] Inc. Free Software Foundation. *GNU General Public License.* Disponível em: http://www.linux.org/info/gnu.html - Último acesso em: 14 fev 2010, 1991.

[22] Luis Eduardo Leyva del Foyo. *Administración de Interrupciones en Sistemas Operativos de Tiempo Real.* PhD thesis, Departamento de Computación. Centro de Investigación de Estudios Avanzados del Insituto Politécnico Nacional, México, 2008.

[23] Philippe Gerum. *Xenomai-Implementing a RTOS emulation framework on GNU/Linux.* Disponível em: http://www.xenomai.org/documentation/branches/v2.4.x/pdf/xenomai.pdf - Último acesso em: 10 jul 2009, 2004.

[24] T. Gleixner and D. Niehaus. Hrtimers and beyond: Transforming the linux time subsystems. In *Proceedings of the Linux Symposium*, Ottawa, Canada, 2006. Proceedings of the Ottawa Linux Symposium, Ottawa, Ontario, Canada.

[25] Luís Henriques. Threaded irqs on linux preempt-rt. *OSPERT 2009*, page 23, 2009.

[26] Arnd C. Heursch, Dirk Grambow, Dirk Roedel, and Helmut Rzehak. Time-critical tasks in linux 2.6: Concepts to increase the preemptability of the linux kernel. In *Linux Automation Konferenz*, Germany, 2004. Citeseer.

[27] Intel. Ia-pc hpet (high precision event timers) specification. Technical report, Intel Corporation, 2004.

[28] Dongwook Kang, Woojoong Lee, and Chanik Park. Kernel thread scheduling in real-time linux for wearable computers. *ETRI journal*, pages 270–280, 2007.

[29] Jane W. S. Liu. *Real-Time Systems*. Prentice-Hall Inc., New Jersey, 2000.

[30] Robert Love. *Linux Kernel Development*. Novell, 2005.

[31] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wiley Publishing, Inc., Indianapolis, 2008.

[32] Ingo Molnar. *PREEMPT-RT*. Disponível em: http://www.kernel.org/pub/linux/kernel/projects/rt - Último acesso em: 25 jan 2010, 2005.

[33] Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. 30 seconds is not enough!: a study of operating system timer usage. In *EuroSys*, pages 205–218, Glasgow, Scotland, 2008. ACM New York, NY, USA. Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008.

[34] Carlos Alexandre Piccioni, Cássia Yuri Tatibana, and Rômulo Silva de Oliveira. Trabalhando com o tempo real em aplicações sobre o linux. Technical report, Universidade Federal de Santa Catarina, CTC - Centro Tecnológico, DAS - Departamento de Automação e Sistemas, 2001.

[35] POSIX.13. Information technology -standardized application environment profile-posix realtime application support (aep). IEEE Std. 1003.13-1998, 1998.

[36] Steven Rostedt and Darren V. Hart. Internals of the rt patch. In *Proceedings of the Linux Symposium*, volume 2007, pages 161–172, 2007.

[37] David A. Rusling. The linux kernel. *The Linux Documentation Project*, 1996.

[38] Ítalo Campos de Melo Silva, Rômulo Silva de Oliveira, and Luciano Porto Barreto. Método para diminuir o tempo de interferência de tarefas de tempo real. *Workshop de Tempo Real e Sistemas Embarcados - Sessão WIP*, 2010.

[39] John Stultz, Nishanth Aravamudan, and Darren Hart. We are not getting any younger: A new approach to time and timers. In *Linux and Open Source Conference*. Proceedings of the Linux Symposium, 2005.

[40] Linus Torvalds. *Linux Kernel Version 2.6.31.6.* Disponível em: http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.31.6.tar.bz2 - Último acesso em: 20 dez 2009, 2009.

[41] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. General-purpose timing: The failure of periodic timers. Technical report, Technical Report 2005-6, Hebrew University, 2005.

[42] Santiago Urueña, José Pulido, José Redondo, and Juan Zamorano. Implementing the new ada 2005 real-time features on a bare board kernel. In *Proceedings of the 13th International Real-Time Ada Workshop (IRTAW 2007)*, page 66. ACM, 2007.

[43] Matthew Wilcox. I'll do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers. In *Linux.Conf.Au*, 2003.

[44] Albert S. Woodhull and Andrew S. Tanenbaum. Sistemas operacionais: Projeto e implementação. 3 ed., 2008.